[美] Kevin Hoffman Dan Nemeth 著
TalkingData：宋净超 吴迎松 徐蓓 马超 译

Pearson
Broadview
www.broadview.com.cn

Cloud Native Go

构建基于Go和React的云原生Web应用与微服务

Cloud Native Go:
Building Web Applications and Microservices for the Cloud with Go and React

中国工信出版集团
電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

[美] Kevin Hoffman Dan Nemeth 著
TalkingData：宋净超 吴迎松 徐蓓 马超 译

Cloud Native Go

构建基于Go和React的云原生Web应用与微服务

Cloud Native Go:
Building Web Applications and Microservices for the Cloud with Go and React

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书旨在向开发人员展示如何构建适用于大流量、高并发场景下的云原生 Web 应用。本书从搭建开发测试环境开始，逐步介绍使用 Go 语言构建微服务的方法，通过引入 CI/CD 流程和 Wercker、Docker 等工具将应用推送到云中。结合微服务构建中的后端服务、数据服务、事件溯源和 CQRS 模式、基于 React 和 Flux 的 UI 设计等，本书最后构建了一个基于 Web 的 RPG 游戏 *World of FluxCraft*，可以作为使用 Go 构建云原生 Web 应用的参考，适合于云计算与 Go 语言编程从业者们阅读。

版权贸易合同登记号　图字：01-2017-4041

图书在版编目（CIP）数据

Cloud Native Go：构建基于 Go 和 React 的云原生 Web 应用与微服务 /（美）凯文·霍夫曼（Kevin Hoffman），（美）丹·内梅斯（Dan Nemeth）著；宋净超等译. —北京：电子工业出版社，2017.7

书名原文：Cloud Native Go: Building Web Applications and Microservices for the Cloud with Go and React

ISBN 978-7-121-32109-2

Ⅰ. ①C… Ⅱ. ①凯… ②丹… ③宋… Ⅲ. ①程序语言－程序设计 Ⅳ. ①TP312

中国版本图书馆 CIP 数据核字(2017)第 159584 号

策划编辑：孙奇俏
责任编辑：徐津平
印　　刷：三河市鑫金马印装有限公司
装　　订：三河市鑫金马印装有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编：100036
开　　本：787×980　1/16　印张：16.5　字数：310 千字
版　　次：2017 年 7 月第 1 版
印　　次：2017 年 7 月第 1 次印刷
定　　价：69.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：010-51260888-819，faq@phei.com.cn。

推荐语

云原生是一种新型的架构设计模式和业务理念。云原生使得业务系统可以规避物理资源的限制而享受云的弹性，还帮助开发者用模块化的方式快速构建了松耦合的业务系统。而 Go 语言则因其分布式友善性和高效性被广泛应用于如 Docker、Kubernetes 等流行的云原生开源项目中，成为了云计算从业人员的宠儿。本书将两者融合，既有架构层面的最佳实践，又有从头搭建真实应用的实战演练。而译者更是用流畅的文字，将这本理论结合实践的读物呈现在了国内读者面前。

才云科技 CEO，张鑫

随着 Cloud 的普及，应用程序的架构也需要适应趋势而有所改变。于是，Cloud Native Application 的概念被提出来了。虽然当前云原生应用还在不断演化中，具体会是什么样子没有一个定论，但这本书通过一些有趣的示例，可以让大家体验到应用架构和开发思路与以往相比的不同。由于本书中的示例是用 Go 语言描述的，所以本书也可以作为 Go 语言云原生应用开发的案例教程，相信大家可以从中学到许多。

QingCloud 架构师，王渊命

几乎每个程序员都爱 Go 语言，也都爱云原生开发。如果能用属于云时代的编程语言 Go 来开发云原生应用程序，那一定是一件很酷的事情。这本书告诉我们，这件很酷的事情完全可以成为现实。无论云原生的工具链、微服务的构建方式，还是中间件与数据库，乃至前端视图开发，这本书中都给出了实践准则与详细示例。如果你是一位热爱 Go 语言编程的程序员，那么就拿起这本书开始一场愉快的实战之旅吧！

网易云基础服务总经理，陈谔

这本书是一本写给云时代开发者的书。什么是 Cloud Native 应用？应用开发者该如何开发一个更适合在云上运行的应用程序？这些问题是任何一个云计算从业者，尤其是云应用开发者无法回避的问题。本书循序渐进地向我们展示了如何利用 Go 语言实现微服务、持续部署、ES/CQRS 模型等云时代应用开发的标签性技术，Cloud Native 概念下的云应用形态也随之轮廓毕现、展露无遗！

BoCloud 博云 CTO，李亚琼

目前，Go 语言在云计算领域的地位类似于 C 语言在操作系统层面上的地位。本书是一本面向实战的技术书。作者选用了一整套技术栈、技术理念甚至哲学，带领读者逐步踏入基于 Go 语言的云原生应用程序开发的世界。我亲自运用过书中所讲的大部分技术，也很认同作者的主要观点。如果你也正在开发云上的应用项目，相信这本书可以作为你的重要参考。

《Go 并发编程实战》作者、GoHackers 社群发起人，郝林

伴随着如 Docker、Kubernetes、etcd、InfluxDB 等诸多优秀云计算开源项目的成功，Go 语言也逐渐成为当今最“火”的语言之一，其简单、并发性好、高效等特性获得了越来越多的公司及个人的青睐。另外，云计算逐渐成为业界的潮流和趋势，那么如何能找到一种全面拥抱云构建应用程序的方法呢？本书列举大量 Go 语言示例，结合构建云原生应用所需要遵循的“道”，很好地为大家解答了这个问题。

腾讯云 PaaS 产品总监，邹辉

Go 语言被誉为云时代的系统语言，而目前市面上刚好缺少一本基于云平台的实战经验手册，而这本书的出现恰好弥补了这个空缺。本书详细地讲解了在云计算时代如何使用 Go 语言进行应用程序开发、自动化测试、运维及部署，我非常看好这本书，它一定能成为云计算时代的 Go 语言开发标准手册。

Apple 核心系统高级工程师，谢孟军

Go 语言具备简洁的语法、超高的开发效率以及优异的性能，这使其成为云计算时代后端开发的首选语言。这本书不是一本关于 Go 语言的教程，而是从实战的角度出发，介绍如何在云环境下以 Go 语言为核心开发业务系统的实践手册，书中还介绍了系统开发过程中需要遵循的开发原则和哲学，相信读者会从中获益匪浅。

PingCAP 工程副总裁、TiDB 技术负责人，申砾

拿到本书的英文原版书籍时，我快速浏览了内容，当即就决定组织团队翻译这本书。因为在这本书中我看到了整个技术运营部门过去两年走过的路，以及未来要走的路，也看到了 Go 开发的基础设施和微服务将是未来松耦合和弹性架构的重要支持者。这本书包含 Go 语言基础知识、持续交付、Web 框架、微服务以及安全等整个云服务开发流程中的关键点，是一本不错的实战手册。再次感谢这本书的翻译者宋净超、吴迎松、徐蓓、马超。

TalkingData 运维总监，潘松柏

推荐序 1

作为一个在 IT 行业摸爬滚打 20 余年的老程序员，我一直认为程序员的工作不仅仅是进行代码编写。很多情况下，程序员的工作和作家类似，都是在进行创作。很多非常出色的程序员同时也是出色的作家，比如程序员王小波除了是一个 C 语言和汇编语言高手，同时还创作了《时代三部曲》，进而成为知名的作家。当然，大部分程序员没有王小波那份驾驭文字的功力，没办法跨界写小说。不过，总结日常的一些技术点滴使其成为文章或者翻译国外的博客、技术书籍，这些对于大部分程序员来讲还是不难做到的，因此这也成为我们团队所有程序员的 OKR。

最近几个月的周六，我经常看到宋净超同学静坐在自己的工位上，或若有所思，或埋头打字，屏幕上闪烁的不是编程的 IDE，倒像是码字的 word 文档。经过几个月的辛苦努力，终于，由宋净超、吴迎松、徐蓓、马超几位技术运营团队的同学翻译的这本《Cloud Native Go：构建基于 Go 和 React 的云原生 Web 应用与微服务》摆在了我的面前。

Cloud Native 的概念来自 Pivotal 的 Matt Stine，是面向现代 DevOps、微服务、持续集成等技术的一种思想，其本身并不是某一种具体的技术。顾名思义，这本书就是将这种思想利用 Go 语言进行实践和落地。结合我们技术运营团队过去两年的工作，我深刻地理解他们为什么要翻译这本书，因为这本书中的很多思想正是我们技术运营团队在过去两年中一直践行的。对于在实践的路上苦苦前行的程序员来讲，看到一本与自己的技术理念非常一致的书，内心的激动可想而知，我相信这也是他们要加班加点将这本书翻译出来并介绍给国内广大同行的重要原因。

从一个经常阅读技术书籍的读者角度来看，本书是一本很贴近实战的技术书籍。对于没有 Go 语言开发经验的读者来讲，这本书介绍了 Go 语言的基础知识，并且指导读者去实践，从而为掌握书中的其他内容打下基础。不过，这毕竟不是一本 Go 语

言专著，如果想了解更多 Go 语言的特性和高级用法，还需要查阅专门讲解 Go 语言的书籍或访问 Go 语言技术社区。既然这本书是面向 Cloud Native 的，那么关于 Cloud Native 的概念无疑才是本书的核心内容。在这本书中，大家除了可以了解到持续交付、测试优先、微服务、服务治理、数据服务、CQRS、云安全等概念，同时还可以通过书中的示例一步步地实践，最终完成一个真实的 *World of FluxCraft* 项目。一本优秀的技术书籍应该能够在理念上给人以启迪，让人产生思考和共鸣，同时又能够真正落到实地，让技术人员可以亲自去探索和验证，而这本书无疑是优秀书籍的代表。

工作多年，我阅读过很多由国内技术人员翻译的计算机方面的书籍，体验不尽相同。很多技术书籍的翻译水准欠佳，且不说能否达到信、达、雅的境界，能够满足没有常识性错误和语句通顺这两个基本要求就已经很难得了，因此，有的时候我宁愿去读英文原版。刚刚拿到这本书的翻译稿时，老实讲，我十分担心翻译质量，在这个 AI 逐渐取代人类工作的时代，如果翻译质量不高，不如使用谷歌翻译更为合适。然而通篇读下来，整本书的翻译质量出乎我的意料！虽然是翻译稿，但是整本书的语言风格非常本土化，并且能看出译者的文字功底非常深厚。无数个周末的无休，只是为了能够给大家带来一部高质量的技术书籍，也不枉原作者辛苦创作，我觉得技术运营团队的同学们的这种态度非常难得！

由于负责 TalkingData 的主要线上业务和数据业务的技术研发工作，因此我每天都会面临着如何能够使线上系统在并发压力和数据规模持续增加的情况下还能保证稳定和快速迭代的挑战。为了应对这些挑战，我们在几年前就开始尝试将 DevOps 的理念引入团队中，开发并开源监控报警系统 OWL，这让我们能够全天候、多通道地支持系统报警。同时我们也将微服务化、灰度上线、端到端自动测试等应用于日常工作中。

这本书中的很多理念给了我深刻启发，也让我坚信我们正走在一条正确的路上。相信国内和我们面临同样挑战的团队还有很多，这本书无疑可以给面临类似问题的团队带来很大的帮助。同时，我也衷心希望我们的技术团队未来能够将自己的经验和教训积累下来，出版我们自己的原创技术书籍。

阎志涛
TalkingData 副总裁
2016 年 6 月

推荐序 2

很多年以前，我便听说过 Go 语言的大名，因为它的创始人中有大名鼎鼎的 Ken Thompson 以及 Rob Pike。年轻一代的程序员或许不了解这两位“爷爷”辈的程序员，但在我学习计算机的年代，这两位大师非常受人崇敬，他们参与开发的 Unix、Plan 9、UTF-8 等也都是可以载入史册的伟大产品。尤其是 Thompson，他早在 1983 年就因对 Unix 以及 C 语言做出卓越贡献而获得了图灵奖。不过当我第一次听到 Go 这个新的程序语言时，多少还是有些怀疑，Go 究竟能不能被程序员所接受呢？毕竟现在的开发者可以选择的语言工具已经极其丰富，无论在哪一种场景下，都已经存在太多的选择。

几年过去了，随着 Docker 的大热，我才突然意识到其背后的开发语言竟然就是这个新生不久的 Go。随之而来的是，越来越多的企业和产品开发项目开始采用 Go。这个名单很长，其中包括 Kubernetes、OpenShift、CoreOS、MongoDB、Twitch 以及 Uber 等。尤其让我感到惊讶的是，以全面采用 Python 语言著称的 Dropbox 居然也将核心的组件从 Python 迁移到了 Go 上面，原因在于程序语言的性能不同。受到这个事件的影响，我开始将 Go 语言加入到我的学习清单里面。

众所周知，云计算已经成为了这个时代中 IT 技术发展最重要的方向，同时因为我所任职的企业 AWS 在云计算领域拥有巨大的影响力，于是我就会特别留心一切与云计算开发相关的话题。自从 2015 年 AWS 发布了针对 Go 语言的 SDK，越来越多的开发者开始了 Go 语言的云计算开发之旅。我相信许多开发者在学习的过程中需要的不仅仅是一门讲解程序语言语法的教程，他们更希望的是拥有一本针对云计算的 Go 实践开发手册。《Cloud Native Go：构建基于 Go 和 React 的云原生 Web 应用与微服务》应该就是这样的一本书，书中的一切内容都围绕着云计算的实践来展开，当

中每一段代码示例都可以被应用到实践中。

写一本书是很辛苦的，而高质量的翻译无疑是使这样一本书能够被广泛接受的关键所在。感谢作者和译者们为此而付出的辛苦努力，也希望所有开发者能够在云计算的时代因此而受益。

古人云：理无专在，而学无止境也。是以为记。

费良宏

AWS 首席布道师

2017 年 6 月

译者序

Go 语言起源于 Google，集中进入大家视线是由于一款革命性的产品——Docker 的发布。从 Go 开始流行，直到其位列 TIOBE 榜单的前 20 名，我们都一直关注着这门语言的发展。近两年来，一批批优秀的基于 Go 语言的开源软件涌现出来，例如 etcd、Kubernetes、Prometheus 等，这些开源软件被广泛应用于我们的生产环境中。

因为 Go 语言非常简洁且功能强大，加之其能够充分利用系统的多个核心组件，实现高性能的网络服务，因此我们于 2014 年将 Go 语言引入了我们的公司 TalkingData，并使用它构建了自己的开源监控系统——OWL（https://github.com/TalkingData/owl），也基于它实现了对 Hadoop 集群虚拟化的探索 magpie（https://github.com/rootosngjc/ magpie）。

刚开始接触到本书时，我们看到书中提到的那些熟悉的技术和理念后感到非常兴奋，通过这本书，我们可以有机会用一种体系化、结构化的方式与大家交流书中的知识和技术。因此我们决定翻译这本书，并且牺牲了许多工作之余的时间来研究书中的内容和细节，除了出于对技术分享始终保有热情，更是因为想要让各位读者早日看到这本关于 Go 语言和云原生技术的好书。

Cloud Native Go 是一本很好的云原生应用实践手册，全书基于微服务理念编写，书中有丰富的示例和代码，这些代码托管在 GitHub 上，读者可以很轻松地获取到。另外，书中还介绍了很多不错的工具的最佳实践，这些工具都是免费的，不需要绑定信用卡，大家可以放心使用。而且，原作者风趣幽默的行文非常有吸引力，不会让大家阅读时感到乏味。

这本书涉及的内容非常广泛，读完本书，读者会对云原生应用的构建规则、微服务划分、测试驱动开发、CQRS 和事件溯源、持续发布流程、安全、故障排查等整

个软件开发生命周期中的重要环节有一个较好的了解，也会对前端开发、前端框架、UI 设计有一定的认识，从而在实际的程序开发过程中更加得心应手。

当然，这本书不是一本专门讲解 Go 语言的书籍，它适用于有一定 Go 语言基础的读者。阅读关于创建 Web 应用的部分时，还需要读者对 Web 应用开发流程有所了解，这对于长期从事后端开发的读者来说可能会比较困难。

除我以外，还有三位 TalkingData 的同事徐蓓、马超、吴迎松参与了本书的翻译。其中，徐蓓翻译了 1~5 章，马超翻译了 6~8 章，吴迎松翻译了 9~12 章，其余的章节由我翻译，同时我也承担了全书译文的审校工作。

本书能够顺利出版并及时与读者见面，要感谢很多人的帮助。感谢公司的大力支持；感谢电子工业出版社的编辑孙奇俏对本书的大力协助和专业指导；感谢在本书的翻译过程中所有通过朋友圈和 https://rootsongjc.github.io/cloud-native-go/网站关注和支持我们的朋友们；最后再次感谢所有译者。希望每一位读者都能从本书中获得想要的知识，希望你们喜欢这本书，衷心感谢大家！

宋净超

2017 年 6 月

前言

当 Dan 和我开始写这本书时，我们不希望它成为一本参考书或“一本语法书”。相反，我们希望能够充分利用自己为 Pivotal 客户构建云原生解决方案的经验，以及近一生的综合经验来为各种规模、形态和行业的公司构建软件。

这本书从一个哲学章节“*云之道*”开始，因为我们坚信构建良好软件的秘诀在于开发人员的心态和纪律，而不是工具或语言。

在本书中，我们将按照测试驱动和高度自动化的方式逐步实现*云之道*，通过一系列章节提高大家在 Go 中构建云原生服务的能力。本书涵盖构建服务的基本原理，中间件技术，Git、Docker 和 Wercker 等工具的使用，还包括云基础设施的相关内容，如基于环境的配置、服务发现以及基于响应和推送式的应用程序。本书涵盖了事件溯源和 CQRS 等模式，书中的所有内容组合成最终的示例，相信可以为大家的项目构建提供灵感。

我们始终秉承着一个坚定的信念——构建软件应该像使用它一样有趣（或更有趣）。如果没有乐趣，那么一定是你做错了。我们希望在使用 Go 构建服务时获得的快乐可以感染读者，希望你在阅读本书时能像我们在写作它时一样，获得更多的乐趣。

关于作者

Kevin Hoffman 通过现代化和以多种不同语言构建云原生服务的方式帮助企业将其应用程序引入云端。他 10 岁时开始编程，在重新组装的 Commodore VIC-20 上自习 BASIC。从那时起，他已经沉迷于构建软件，并花了很多时间学习语言、框架和模式。他已经构建了从遥控摄影无人机、仿生性安全系统、超低延迟金融应用程序到移动应用程序等一系列软件。他在构建需要与 Pivotal Cloud Foundry 配合使用的自定义组件时爱上了 Go 语言。

Kevin 是流行的系列幻想书 *The Sigilord Chronicles*（http://amzn.to/2fc8iES）的作者，他热切地期盼着最终能够将自己对构建软件的热爱与对构建幻想世界的热爱结合起来。

Dan Nemeth 目前在 Pivotal 担任咨询解决方案架构师，负责支持 Pivotal Cloud Foundry。他从 Commodore 64 开始就一直在开发软件，从 1995 年开始从事专业编码，使用 ANSI C 编写了用于本地 ISP 的 CGI 脚本。从那时起，他职业生涯的大部分时间都是作为独立顾问为金融、制药等各个行业提供解决方案，其间不断使用当时流行的各种语言和框架。Dan 最近接受了 Go 作为自己的“归宿”，其间不断热情地将它用于所有的项目。

如果你发现 Dan 没在电脑前，他很可能就是在靠近安纳波利斯的水域玩帆船或飞钓。

致谢

这本书能够诞生要感谢我的家人，特别是我的妻子，她给予了我无限的耐心。尽管我在过去曾多次说过，我不会再写技术书，但事实上目前我正在写另一本技术书。她们忍受了漫长的夜晚，忍受我在家里的地板上迁思回虑，以及为了保证这本书的质量带来的巨大时间消耗。完成这本书比过去完成任何其他工作都让我感到更加自豪，这本书是家人、朋友和杰出的合著者给我的宝贵支持的结晶。

——Kevin Hoffman

❖

这本书献给 A-Team：四个在 Pivotal 工作的人。他们现在正在寻找需要指导的开发人员。如果你需要将软件迁移到云上，他们一定会找到你。

没有这些勇敢的人，编写软件的过程将变得非常无聊和难以忍受，也许永远不会有这本书。事实上，作者们可能已经放弃了他们一直以来所致力的云服务，而是希望余生能在咖啡店当一名咖啡师。

A-Team 成员有：

Dan “Hannibal” Nemeth

Chris “Murdock” Umbel

Tom “Face” Collings

Kevin “B.A.” Hoffman

❖

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **提交勘误**：您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动**：在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/32109

目录

1 云之道

生大材，不遇其时，其势定衰。生平庸，不化其势，其性定弱。

——老子

正如前言中提到的，本书的目的是教大家如何构建云原生（cloud native）应用程序。然而 *cloud native* 这个词，本身带有很多我们想要摒弃的既有观念，例如明确地关注应用而不是人和设计哲学，并且依赖于原始的“12 要素法则”[1]来定义和描绘应用程序。

虽然以应用为中心的设计原则肯定是有价值的，但 cloud native 的文化是发源于构建和设计应用程序的这群人之中的。如果这些人接受了正确的设计哲学，那么他们显然可以更优雅地构建应用程序。艺术家的作品体现出了其在创作时的激情，而 Web 应用程序和微服务也应如此。

构建云应用程序不仅要学习新的库或编程语言，还涉及掌握新的学科、建立和培养新的习惯，并要以不同的方式看待世界。汉字“道”有许多解释，但用于哲学时，它表示方式或途径，尤指一个人做事的方式和途径，例如生活方式或构建软件的方式。为了形容我们所信仰的 cloud native 开发和架构哲学，我们将其称之为云之道。

本章将介绍云之道，这种方式有许多典型的优点。本书中构建的一切，从文字到代码再到支持站点，都流露出我们对道的热爱。希望大家在读完本书后，也能对道有和我们一样的感受。

接下来将讨论以下内容。

- 云之道的优点。

1　12 要素法则，出自 Heroku 最新出版的关于构建云端应用程序的指南，链接为 http://12factor.net.

- 选择使用 Go 语言开发云端微服务的理由。

云之道的优点

与人们普遍的看法相反，我们所做的事并不都是很特别的。有些事情甚至会磨灭我们的激情，成为一种烦恼。我们努力工作，经历添加功能、部署、祈祷它们正常运行的艰难时刻，却忽视了自己构建的是艺术品的事实，不再给它们注入应有的爱和热情。

在某种程度上，这些事会在所有人职业生涯的某些时刻发生。我们期盼出现一位鼓舞人心的老板、一种新的技术或语言甚至一个新的工作来扭转这一切，将我们从深渊中解救出来。

对我们来说，这种解救便是云之道。它改变了我们对软件开发的看法，以及看待世界的方式。我们又可以感受到自己所构建的艺术，软件开发再次成为乐趣，开发者绝不会再用以前的方式来构建应用程序。

纵观软件开发阶段，不管是出于商业目的、创业或是兴趣爱好，或是介于三者之间，其实都已经有一整套既定的准则，遵循它就可以在最大程度上创建云上的可扩展、可靠、可预期的软件。

本书是关于 *cloud native* 软件开发的，所以在整本书中，所生成的每个代码示例和讨论的每个主题都将会具有下文提到的优点，当然这些优点并不是只适用于云软件。事实上，从文字、代码到支持网站，我们在完成这本书的过程中都遵循这些规范。

遵循简单

> **云之道**
> 所做的任何事都要简单化。

质疑一切事物似乎违背简单性。但当所构建的一切都已相当复杂时，绝对没有必要再为工作引入额外的复杂性流程。

质疑每个工具。考虑这个工具是可以简化系统，还是会在系统其他地方引入额外复杂性。如果答案是后者，那么就抛弃这个工具，忽略使用它的原因。

质疑所有的代码。如果它过于复杂以至于无法阅读，请替换造成这种复杂性的编程语言或框架。如果在代码背后有很多隐藏的“戏法”，无法辨别出它会在何时何地如何发生，那么请修改代码。

以下是检验简单性的测试。

- IDE 是否可选？
- 能否通过命令行构建和部署？
- 团队的新成员能否快速理解代码？

工具和 IDE 必须可以自动化地执行例行任务、减少阻碍或时间来简化手动任务，从而使我们能够更好地工作和生活。工具绝不能是强制性的，如果代码必须用特定的 IDE 才能生成或者编译，那绝对不是在秉承*云之道*，也没有在遵循简单性原则。

任何可以通过命令行完成的工作，都可以使用脚本或者持续集成工具自动化地完成。因此，如果能使用命令行来构建、测试和部署应用程序，那么就可以自动化执行所有这些任务。

这似乎是一个尖锐的观点，大家有权反对。然而，至少在读完这本书之前，可以尝试在所有开发工作中遵循这些准则，它一定不会令你感到失望。

那些声称无须使用云端或无须遵循简单性的人都将对他们创造的或不愿摒弃的复杂性而感到羞愧。

测试优先，测试一切

> **云之道**
> *采用测试驱动进行开发。测试一切，处处测试。*

测试是抵御程序偏离期望方式而运行的首要且最好的方法。

几乎每个人都赞同进行测试，但在应该进行什么程度的测试这一点上，很少有团队能达成一致。在解答这个问题之前请先思考：为什么需要测试软件？

当然，我们想要编写完美的软件，希望客户满意，想通过软件赚取大量金钱。但是，测试的核心不是这些。在根本层面上，测试可以给我们信心。

你是否经历过这种时刻：在某种场合下，有人要在一些重要的和有影响力的人面前展示你的应用程序？你是否还记得在关键功能测试之前感受到的恐慌？

恐惧

缺少测试是产生恐惧的根本原因。

感到恐惧是因为担心出现以下情况：应用程序执行的不确定性会影响到整个系统，导致难以估量的压力、系统故障，最坏的情况是在生产系统中出现客户可感知的问题。

我们需要做的是树立信心，以信心取代恐惧。对系统的信心是不断累积起来的，它始于最小的可测试单元。从那一刻开始，随着代码库的扩展，信心不断建立，并通过测试不断完善。

图 1.1 进行了详细说明。一个类通过了完全的单元测试，我们就对这个类树立了完全的信心，接着对库中所有的类进行单元测试，这样就对整个库树立了完全的信心。相反地，如果无法信任应用程序的构建块，那么对整个应用程序的信心就会降低。从类、库到服务，如果缺乏测试，我们的信心就会呈指数下降。

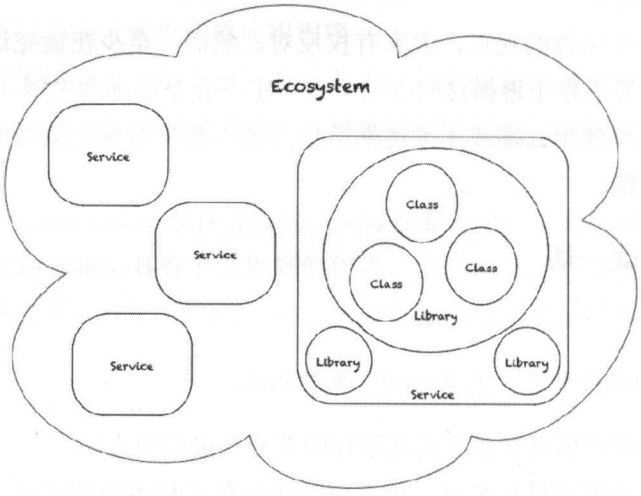

图 1.1　在微服务生态圈中测试信心的区域

只有完全信任构成这个代码库的所有库类，才能对整个服务树立信心。最后，只有当服务生态系统内的每个服务都经过充分测试（无论是内部还是多个服务边界间的集成测试）时，我们才能信任整个服务生态系统。

大家可能经常会遇到这类情况：团队负责人认为不值得在测试上耗费大量时间。进行测试会产生很大的开销，这很容易通过电子表格计算。为了反驳这种观点，可

以参考以下的案例。

当在本地构建应用程序时，可以使用所有能想到的工具。可以连接调试器，可以设置和命中断点。在某些情况下，甚至可以暂停一个应用程序，操纵内存中的数据，然后恢复。这给了我们产生信心的错觉，在调试器上花费的时间使得集成和单元测试被认为是不必要的。

再考虑以下场景：我们正在构建的不是软件的一小部分，并且软件部署在离我们只有几英尺的电脑中。想象一下，假如我们实际上构建的是一艘将要发射到外太空的太空船，一旦启动，就无法再接触它，无法抓住螺丝刀再进行最后一刻的调整。如果在远离基地的几百万英里外，如某些部件发生灾难性的故障，那么太空船的整个旅程就将结束。

在云端部署时，无法进行大量的手工控制。它虽不像发射卫星那么复杂，但是失去了对实例的实际控制，不能设置断点，通常也无法进行运行时自我检查。

如果下次再有人挥舞电子表格，声称测试成本太高，我们只需回复和询问他们产品在每个阶段构建完成与完全失败的成本，因为这是测试的实际成本。

回到信心层面，如果对程序正常运行充满信心，那么就可以将这种信心作为发挥其他优势的基石，如持续交付（下面讨论）。

这一切的前提是，我们应该采用测试驱动开发[1]。必须测试所构建的每个服务的一切内容，不管是内部还是外部的，以此建立对服务的信心。

尽早发布，频繁发布

> **云之道**
> *将每次代码提交都当作潜在的生产发布，并通过持续交付流水线进行部署。*

在上一节中，我们讨论了那些使用我们开发的应用程序的人的恐惧，这往往是我们缺乏对应用程序正常运行的信心所导致。接下来还需要面对另一种恐惧：发布的恐惧。

有些公司按照发布世界一流活动的标准制定发布规范。他们提前几个月开始计

1 想了解更多关于测试驱动开发的信息，请查阅 Beck, Kent. (2003) *Test-Driven Development: By Example*. Addison-Wesley Professional.

划，急救人员处于待命状态，救护车和消防队员也全部就绪，几乎每一个人，哪怕仅检查过一行代码或监督代码提交的人，都在午夜时守在电话旁，执行应用发布。仿佛吉尼斯世界纪录丛书就摆在电话旁，以便纪录历史上发生的第一次成功发布。

采用这种方式的公司会尝试在发布时投入更多的资源、更多的人、更多的基础设施和控制来缓解出现的问题。有时甚至会推迟发布，降低发布频率。这其实恰恰与应该做的完全相反。

克服发布恐惧的唯一方式是更频繁地发布。

要使团队中的每名开发人员都坚信，每次对源码管理系统的提交都会在几天之内到达生产系统，这会产生一连串的好处。首先，这样能培养严格性和纪律性，这是大多数人的代码中所缺少的。此外，如果知道代码会被快速发布，那么开发者更有可能会测试代码，因为不想对生产系统的发布失去信心，不是吗？

自动化一切

> **云之道**
> *所有可以自动化的，都应被自动化。*

现在我们充满信心，因为代码已经通过测试，并以有利于测试的方式被编写。即使频繁地发布，也不再担心时刻处于发布的阴影下。我们开始更好地生活，同时也提高了生产力。

任何每天做的超过一次的事情，都适合自动化。

发布本身是最重要和最需要被自动化的流程。在代码提交的几分钟之内，一些自动化系统应该测试代码，验证代码是否符合标准，并将一个稳定的构建工件部署到环境中，以用于手动和集成测试。

我们以手工方式执行的操作很容易出错。容易分散注意力、忘记执行步骤，可能会按照错误的顺序执行操作，也可能会引入额外的不必要的步骤。一旦确定一个自动化流程，它就能为我们树立更多的信心。

应该做到这一点：在向源码管理系统提交代码后，警铃没有大作，黑武士的头像没有闪亮，充满愤怒的邮件没有发送，我们由此树立信心——变更没有导致单元或集成测试失败。当然还有更多的测试需要在产品上线之前完成，但是我们对即将通过准生产验证流程的发布提交充满信心。

综上所述，关于自动化的最终建议如下。

流程中任何时常重复的部分，如果不能被按钮或者脚本代替，那么就属于过于复杂、脆弱或两者兼有的部分。

只有当拥抱自动化，并可以自动将代码提交到云端时，才能真正开始从云端开发中受益，并从构建单一的微服务扩展到构建微服务生态系统。

建立服务生态系统

> **云之道**
> 任何事物都是服务，包括应用。

在讨论生态系统之前，我们要先发泄一下对微服务[1]的不满。与任何流行语一样，它已被过度使用和注水。我们坚信所有服务都应该是微服务，所以前缀“micro”是完全不必要的。

多年以来，我们喜欢构建庞然大物。即使在“n-tier”或“3-tier”应用程序流行之时，这些 n-tier 应用程序仍然是一体式的，它们只是更具组织化而已。

在单体式应用程序中，系统的每个关注点和功能性的需求都包含在一个庞大的结构中，这违背了云的基本原则：简单、易于自动化和易于发布。

在单体中，每次变更都需要发布整个应用程序，这促使我们想要极力避免“all hands”的发布方式。这样的应用程序很难维护，它的启动和停止速度很慢，而且依赖关系紧密耦合，通常很难部署到云端。

微服务只是遵循了**单一责任原则**（SRP）这一松散定义的服务。SRP 源自面向对象的设计模式，即一个服务只负责一个功能。

应用程序也只是具有一个或多个呈现 GUI（例如 HTML）的微服务而已。

全书正是基于微服务这一概念编写的，并且讨论了什么应该是服务，而什么不是，以及如何将已有的单体应用程序切割成粒度更小的服务。

云的优势在于，我们无须构建巨型的、单体的应用程序，取而代之的是构建可以在一个生态系统中共存的服务。这有助于养成一些好习惯，例如构建强版本约束

1 想了解更多关于微服务的内容，请查看 Newman,Sam. (2015) *Building Microservices*. O'Reilly Media.

的 RESTful 接口、在发布流程中包含服务交互测试、为未知功能和未知客户提供灵活的支持。

为什么使用 Go

大家已经通过前面的介绍了解了云之道，接下来，你可能会想：为什么使用 Go？是什么使得 Go 成为用于构建云端服务和应用程序的理想语言？

以下三个主要原因使得我们选择 Go 作为构建云端应用程序的首选语言：简单、开源和易于自动化。

简单

Go 的简单性存在一种引人注目的美。表面上，它甚至可以传递 ANSI C 的简化变量，但它也足够强大，以至于可以满足当今最苛刻的软件需求。

这种简单不仅流于表面——Go 避免了不必要的复杂性、额外流程和一些令人讨厌的步骤。Go 不像 Java 或.NET 那样编译生成中间字节码，它直接生成本地二进制文件。因此，这种简单性不是以牺牲什么为代价，它实际上是提高了性能。

当克服了对于 Go 不支持类、纯函数式编程[1]这种最初恐惧，大家最终一定会像我们一样重新爱上编程。

开源

Go 不仅是一种开源语言，它的背后更有社区的拥护和支持。正如接下来我们将在本书中看到的，通过 GitHub 和其他仓库共享开源模块是 Go 语言及其核心工具的首要理念。

易于自动化和 IDE 自由化

正如我们在本章前面提到的，如果可以使用脚本定向控制构建流程，那么就可以将它自动化。我们能使用 Go 实现的任何东西，从编译、执行测试到静态分析，都可以通过一个简单的命令行来完成。

---

1 虽然 Go 支持高阶函数以及将函数作为参数，但是它缺少函数式编程应有的一些特性。这些讨论不在本书的范围内。

使用 Go 的另一好处是可以自由选择文本编辑器。请务必在 cloudnativego.com 上查看关于如何使用 IDE 进行设置和自定义 Go 开发环境的建议。

也可以在相关网站 github.com/cloudnativego/上查看本书中使用的代码文件和其他资料。

本章小结

本章谈到了一些哲学理论，并制定了定义和准则。我们坚信，开发时的心态是决定任何云原生项目成功或失败的主要因素。

这就是在本书一开始就介绍了云之道的原因，而且随着阅读的深入，这种哲学理念会渗透到我们所做的每一个决定、写的每一行代码以及每一个自动化测试中。

本章还谈及了 Go 本身如何善于构建简单、优雅、高效、快速的微服务的特性。Go 看起来不是我们最喜爱的语言，因为它没有奇特的库或可以实现混淆代码的不同方式。但是，Go 确是我们最喜爱的语言，正是因为缺少了这些东西，才使得我们的思想可以毫无阻力地变成代码。

2 开始

获得成功的秘诀是尽早开始。

——马克·吐温

正如我们在本书中提到的，选择一个合适的环境（本地的或自动构建的）是项目成功的关键。如果在工作站中使用了正确的工具并经过正确配置，那么在此之上进行开发会使人神清气爽。相反，一个蹩脚的环境会使开发人员感到窒息。

本章中不会提供任何我们自己编写的代码。相反，我们将关注编写代码所需的最小工具集，以便在下一章编写代码时可以感到简单和轻松。

本章将讨论以下几点。

- 安装 Git，确保有一个 GitHub 账户。
- 安装和配置 Go 命令行工具。
- 测试并验证工作环境。

正确的工具

我的祖父总是告诉我，要时刻确保使用最好的工具。这与传统的建议“总是使用正确的工具”略有不同。我想我的祖父是试图通过这句话传授以下智慧。

首先，如果想走捷径，就不要在工具上走捷径。任何能使生活更简单，使阻力或压力降低的东西，至少需要首先评估。第二，也许是最重要的理念，应该使用最合适的工具。例如，锤子会是蹩脚的螺丝刀，而螺丝刀则是不顺手的扳手，臃肿的 IDE 会生成更多的代码、配置文件，这会让开发人员极度恼火。

我们想要走捷径，这并不意味着反对使用工具，只是反对在使用工具时引入额

外的风险。本书中将使用一些命令行工具和网站，但不会强迫大家使用 IDE。我们坚信，任何强制用户使用特定 IDE 的编程语言或框架都会预置一些代码或增加其他负担。

因此，我们将讨论 Git 命令行工具、Go 命令行工具以及如何验证开发环境。至于在什么环境中编写代码，则由大家自己决定。

配置 Git

这看起来可能很奇怪，我们在本书中要求大家做的第一件事竟然是安装 Git。Go 是一种开源语言，它本身及其提供的工具基本都托管在 GitHub 和开源社区上，这意味着必须安装源代码控制客户端（如 `git`、`mercurial` 和 `bazaar`）来建立 Go 的开发环境。

如果使用 Linux 工作站或虚拟机，那么也可以使用 `apt-get` 完成所有工作，一次性地安装 `git`、`mercurial` 和 `bazaar`。如果使用 Windows，那么建议将系统升级到 Windows 10 anniversary edition，并在 Windows 上安装 Bash（老实说，Windows Go 的开发人员都必须使用它）。

如果使用 Mac，则需要安装 **Homebrew**。

安装 Homebrew

Homebrew 是我们的朋友。作为在 Mac 上工作的开发人员，这很可能是最常用的工具之一。Homebrew 是一种允许安装其他工具的软件，这些工具有很多是必需的。可通过下列方式在 Mac 上安装 Homebrew（以下命令中的任何换行符都仅用于格式化，而不应该被键入）。

```
ruby -e "$(curl -fsSL
  https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

如果没有安装 Ruby，可能是因为没有安装 Xcode 命令行实用程序。这种情况下可以在 shell 中使用以下命令进行安装。

```
xcode-select -install
```

要验证 Homebrew 是否安装成功，可以使用以下命令。

```
brew doctor
```

此命令将检查所安装的 Homebrew。它可能会提示需要拥有一些目录权限，也可

能会提示 Homebrew 已过期。如果是提示已过期，那么请按照指示进行更新。

接下来，安装 Homebrew **Cask**，它自我标识为“Mac 上缺失的应用管理工具”。

```
brew install caskroom/cask/brew-cask
```

通过以上步骤， Homebrew 便安装完成了，现在可以通过它来安装 Go。

安装 Git 客户端

在安装之前，请在 shell 中运行以下命令。

```
$ git -version
```

如果有一个相对较新的版本（本书写作时是 2.8.2），则可以继续安装。注意，如果曾经使用 XCode 在 Mac 或 iOS 上进行过开发，那么可能会有一个 Apple 自带的 Git（会在版本中说明）。如果遇到了 Git 不可用的情况，可以重新使用 Homebrew 进行安装。

```
$ brew install git
```

如果已经安装了 Apple 版本的 Git，并且它的版本比 Homebrew 中的要低，那么可以简单地直接安装 Homebrew 版本的 Git。

安装 Mercurial 和 Bazaar

正如前面提到的，Go 本质上是和开源社区联系在一起的，并且也相应地是和源代码版本控制仓库联系在一起的。GitHub 不是大家感兴趣的第三方 Go 包的唯一来源。如果想确保 Go 环境正常工作，还需要安装两个其他仓库工具：`mercurial` 和 `bazaar`。

只需通过 Homebrew 就可安装以上两个工具，命令如下。

```
$ brew install mercurial
$ brew install bazaar
```

接下来，需要检查是否已经安装了所有的版本控制客户端，通过 Homebrew 的以下命令可以检查安装状态。

```
$ brew info git
$ brew info mercurial
$ brew info bazaar
```

创建 GitHub 账户

稍后在讨论 Go 工作区的设置时会提到，Go 中绝大多数第三方软件包都存储在 GitHub 上，可以通过 GitHub 分享自己的包。简而言之，如果想要使用 Go 完成较复杂的功能，必须要和 GitHub 进行交互。

如果还没有 GitHub 账户，请将网站页面加入书签，并立即获取免费账户。在创建此账户之前，请不要继续前往下一页或执行其他操作。对我们来说，GitHub 就像氧气，没有它我们将无法生存。大家可以在 https://github.com/autodidaddict（Kevin）和 https://github.com/dnem（Dan）上看到我们的 GitHub 账户。

创建 Go 环境

强烈建议大家阅读 Go 官方“入门”页面 https://golang.org/doc/install。根据自己的操作系统和首选项的差异，Go 的安装也会有多种不同的方式。

可以在 Mac 上下载安装程序包，但为了与已经安装的其他工具一致，并以最简单的方式保持工具的最新版本，建议通过 Homebrew 安装 Go。相应地，可以看到类似下面的输出。

```
$ brew install go
==> Downloading
https://homebrew.bintray.com/bottles/go-1.6.2.el_capitan.bottle.tar.gz
######################################################################## 100.0%
==> Pouring go-1.6.2.el_capitan.bottle.tar.gz
==> Caveats
As of go 1.2, a valid GOPATH is required to use the `go get` command:
  https://golang.org/doc/code.html#GOPATH

You may wish to add the GOROOT-based install location to your PATH:
  export PATH=$PATH:/usr/local/opt/go/libexec/bin
==> Summary
    /usr/local/Cellar/go/1.6.2: 5,778 files, 325.3M
```

配置 Go 工作区

当 Go 安装完毕后，需要配置 Go 的工作区。可以选择自己喜欢的任意目录，但它需要具有访问权限。我们编写的所有代码都将放置在这个目录下，下载的所有包都将位于这个目录下的路径中。

设置目录时，请修改配置文件，以便在每次打开终端时，`$GOPATH` 环境变量都

可以指向此目录。

接下来，修改配置文件，以确保$PATH环境变量包含$GOPATH/bin，这样便可以允许我们运行已安装的Go应用程序。

假设已经决定将Go工作区设置为位于主目录下的一个名为Go的目录，用户为goguru，那么需要将$GOPATH环境变量设置为/Users/goguru/Go，并将/Users/goguru/Go/bin添加到$PATH中。

包括已经创建的应用程序或者共享软件在内的所有包，都会存储在$GOPATH/src下的子目录中。例如，我们创建了一个象棋游戏，并且将GitHub账户命名为autodidaddict，那么该游戏代码的根目录将位于$GOPATH/src/github.com/autodidaddict/go-chess中。

Go工作区不是随意组织的，它必须能够正确定位并为构建的所有内容提供依赖项（或下载缓存在本地）。作为Go的初学者，产生问题最多的原因是没有遵循Go工作区组织结构的基本原则。

如果在使用本书中的示例或其他Go代码时发现奇怪的编译失败现象，请仔细检查Go工作区中代码的位置。

检查环境

通常情况下，我们需要在编写代码之前进行测试。一般先运行几个冒烟测试，以确保Go工作区和所有命令行工具可以正常工作。

要做的第一件事是执行go get命令，它从仓库（通常是GitHub）上获取外部依赖，并将其复制到当前Go工作区的相应目录中。

可以通过go get命令运行一个简单的“hello world”程序来执行以上步骤，代码可以通过GitHub获取。

```
$ go get github.com/cloudnativego/hello
```

与Go的风格相似，没有消息就是好消息。执行这个命令后看起来什么都没有发生，但现在我们可以在本地的Go工作区中看到这个包的源目录，并在GitHub上看到相同的文件。

```
$ cd $GOPATH
$ cd src/github.com/cloudnativego/hello
```

```
$ ls
Godeps      Procfile      README.md      buildlocal      main.go
  manifest.yml      wercker.yml
```

到目前为止一切正常。现在，可以开始进行测试了，看看 Go 编译器是否会按照这个目录的内容进行实际构建。

```
$ go build .
```

若一切按照计划进行，应该还是看不到有任何内容输出。如果再次查看这个目录，会看到有一个名为 `hello` 的可执行文件。我们可以像运行其他本地应用程序一样运行它。

```
$ ./hello
[negroni] listening on :8080
```

不需要担心 Negroni 是什么，它看起来已经很像一个 Web 程序，我们可以通过以下命令访问它。

```
$ curl http://localhost:8080
Hello from Go!
```

如果看到“Hello from Go!”，那么证明 Go 工作区已经可以正常工作了，我们已经安装了所有支持 Go 开发的命令行工具和客户端软件。

现在是时候编写一些 Go 代码了！

本章小结

没有人喜欢在书的一整章中都不提及任何代码，但有时这是必要的。在本章中，我们谈到了一些关于工作区设置的命令行工具和配置，它们都是在为本书其他部分的代码编写做准备。

如果没有成功地通过“测试环境”，我们鼓励大家不断尝试去解决它。如果 Go 工作区并未正确设置，那么本书其余部分的任何内容都将无法正常工作。

如果已经准备就绪，那么现在是时候进入 Go 语言最简单优雅的部分了！

3 Go 入门

我喜欢 Go 语言中的很多设计理念，基本上我喜欢它的全部。

——Martin Odersky，Scala 创始人

本章将讲述 Go 语言中的一部分入门知识，也是覆盖贯穿本书的、最常用到的部分，包括以下几点。

- 一个“hello world”示例。
- 介绍函数。
- 通过结构体操作和存储数据。
- 通过结构体使用方法。
- 使用包。
- 创建自己的包。

如果需要深入学习 Go 语言并了解关于它的更多技术细节，建议关注由 Addison-Wesley 出版的 Mark Summerfield 所著的 *Programming in Go: Creating Applications for the 21st Century* 以及 Alan A. A. Donovan 与 Brian W.kernighan[1]合著的 *The Go Programming Language*。

在本章或本书的其他部分，以及其他一些关于 Go 语言的书中，大家都将了解到 Go 并不是一种面向对象的语言。对于这一点的理解越早，将越有助于积累经验。我们认为 Go 语言并不具备真正的面向对象的基本条件，这恰恰是 Go 为了简单性而做出的令人耳目一新的改变。

在第 1 章云之道中提及过的另一件需要注意的事是，Go 崇尚敏捷性和简单性，

1 Brian Kernighan 与 Richie 齐名，在 C 语言领域中做出了卓越贡献。

它的设计不是用来满足时髦语言的一些特性需求的，也不是为了制定规则并在暗中束缚它们。

Go 是一种用于解决实际问题的实践性语言。然而，很多人将 Go 视为一种适合构建小型的、面向命令行应用的特定语言。我们希望通过本书证明，Go 不止可以胜任这些工作，更可以用于构建云原生应用程序和微服务。

本章看起来并没有涵盖 Go 的所有部分，不过不用担心，我们会逐步介绍这些新的概念，本章将主要提供本书其余部分会用到的构建块。

建立 Hello cloud

我们对“hello world”有种复杂的感情。一方面，很多人轻视这类示例，认为它们毫无价值。另一方面，却必须从此处入手，所以打印“hello，world”是很多书中的首个代码示例。

我们将在控制台打印“Hello，cloud”，如代码清单 3.1 所示。

代码清单 3.1　hello-cloud.go

```
package main

import "fmt"

func main() {
     fmt.Println("Hello, cloud!")
     fmt.Println("你好，云!")
}
```

以上的内容相当简单，它是我们可以编写的最基本的 Go 应用程序之一。第一行是声明包（稍后会讨论包），在这个示例中是声明 `main` 包。

然后，我们导入一个名为 `fmt` 的包。这个包提供了基本的格式化函数，有类似 C 的读取、写入和格式化字符串的接口。此代码列表中导入了 `fmt` 包，以便访问 `Println` 函数。

接下来定义一个函数（后面的章节中会讨论），称为 `main`。所有 Go 应用程序都需要一个 `main` 函数作为入口。下一个调用通过 `Println` 打印 *Hello，cloud！* 到标准输出设备。第二个打印函数展示了 Go 的一个小细节，字符串是一个字节数组，然而所有 Go 字符串也可以包含 Unicode 字符（甚至多字节）。讨论 Unicode 编码（在

Go 中称之为 **runes**）[1]已经超出了本章的范围。

要运行此应用程序，只需在 shell 或命令提示符下输入以下命令即可。

```
go run hello-cloud.go
```

若要使此命令正常工作，不必一定使其位于 Go 路径的目录中（在第 2 章“入门”中定义），但是我们应该习惯在 Go 路径中进行工作，甚至是运行最简单的示例。在使用第三方软件包时，这种习惯会很有帮助。

既然这是一个公开的示例，那么便可以从以下路径执行它。

```
$GOPATH/src/github.com/cloudnativego/go-primer.
```

还可以在 Go Playground 中运行此示例，这是一个非常有用的网站，允许快速运行一些 Go 代码，而不会弄乱当前的项目。如果在旅途中，或者无法访问完整的开发环境，那么该网站将是一个快速和方便的替代品（看看我们在那里做了什么）。大家可以通过 http://play.golang.org 访问 Go Playground。

使用基本函数

Go 中的函数可以有零个或多个参数，它们可以返回零个或多个值。除了一些特例，Go 函数的语法看起来非常像经典的 C 语言函数语法。对于那些我们熟悉的可以添加方法的类，这看起来可能有些别扭。Go 中没有类，只有函数，这些函数的可见性由包的范围以及该函数是否可以导出决定。

以下是一个模拟的简单的投掷骰子的函数实现。

```
func dieRoll(size int) int {
        rand.Seed(time.Now().UnixNano())
        return rand.Intn(size) + 1
}
```

此函数名为 `dieRoll`，它接受一个名为 `size` 的整型参数。首先根据当前时间初始化随机函数发生器（需要使用 `rand` 包）。如果使用常数值，那么每次将返回完全相同的骰子点数。接下来，使用 `return` 关键字从函数返回一个值，一个从 1 到骰子点数之间的随机数。

---

2 关于 rune、strings 这些类型的说明请参考网页 https://blog.golang .org/strings 中的内容。

Playground 中的随机化

值得一提的是，Go Playground 使用固定的时钟时间，这意味着随机函数发生器会使用固定的种子，导致我们总是掷到相同的点数。可以通过在本地和 playground 上运行相同的代码来观察这种现象，这会是一个有趣的实验。

如果按照以下方式调用函数，相当于投掷我们最喜爱的龙与地下城的 D20 骰子。

```
dieRoll(20)
```

与大多数编程语言的函数不同，Go 允许返回多个值。但这并不意味着 Go 将类似元组或模式匹配的概念面向对象化了，如下面的函数所示。

```
func rollTwo(size1, size2 int) (int, int) {
        return dieRoll(size1), dieRoll(size2)
}
```

这个函数实际调用了两次 `dieRoll` 函数。从方法签名中可以看出，它返回两个匿名值（即没有为它们分配变量名），两者都是整数。Go 还能声明返回值，如果这样做了，那么在函数的第一行代码执行之前，Go 会将已声明的返回值初始化为它们的默认值。

查看代码清单 3.2 中完整的代码示例，其中定义了很多函数，展示了返回多个值的不同方式。

代码清单 3.2 basic-functions.go

```
package main

import (
        "fmt"
        "math/rand"
        "strconv"
        "time"
)

func dieRoll(size int) int {
        rand.Seed(time.Now().UnixNano())
        return rand.Intn(size) + 1
}

func rollTwo(size1, size2 int) (int, int) {
        return dieRoll(size1), dieRoll(size2)
}
func returnsNamed(input1 string, input2 int) (theResult string, err error) {
```

```
        theResult = "modified " + input1 + ", " + strconv.Itoa(input2)
        return theResult, err
}

func main() {
        fmt.Printf("Rolled a die of size %d, result: %d\n", 6, dieRoll(6))

        res1, res2 := rollTwo(6, 10)
        fmt.Printf("Rolled a pair of dice (%d,%d), results: %d, %d\n", 6, 10, res1, res2)

        named, err := returnsNamed("globule", 42)
        fmt.Printf("Named params returned: '%s', %v\n", named, err)
}
```

以上代码示例缺少了一个重要的部分，就是调用方法并捕获多个返回值的语法。常用的Go模式之一就是定义一个函数，同时返回一个值和一个错误值。如果函数没有出错，错误值就是 `nil`。这种模式在整本书中都有所采用，同样也存在于大多数Go示例或者常遇到的开源项目中。

还需注意的是，返回值的名称只在函数定义的范围内有效，不可以在另一个范围中使用相同的变量名来获取结果。

执行以上程序，输出如下。

```
$ go run basic-functions.go
Rolled a die of size 6, result: 5
Rolled a pair of dice (6,10), results: 1, 6
Named params returned: 'modified globule, 42', <nil>
```

不要将简单与简陋混淆。在第1章中，我们提到Go无须是一个纯函数式编程语言，但这并不意味着它无法实现一些强大的功能。

假设我们想要实现两个不同的掷骰子函数，一个是真实的，一个是虚拟的。然后需要一个包含这些函数的数组，调用时不通过代码就可以知道调用的是哪种函数。在经典的OOP中，可能需要创建一个名为 `IDieRoller` 的接口，然后创建两个实现 `rollDie` 方法的具体类。

但这不是Go的实现方式，我们坚信所有额外的工作都是不必要的。在Go中，可以创建一个类型，将任何匹配签名的函数都当作该类型处理，这个类型可以作为参数传递到函数中，或作为数据存储在结构体或数组中。

首先定义 `dieRollFunc` 类型。

```
type dieRollFunc func(int) int
```

只要这个类型在有效范围内，任何接受一个整数并返回一个整数的函数都可以看作是一个掷骰子函数。现在编写一个函数，使它返回一些其他的函数。

```
func fakeDieRoll(size int) int {
        return 42
}

func getDieRolls() []dieRollFunc {
        return []dieRollFunc{
                dieRoll,
                fakeDieRoll,
        }
}
```

一个非常重要的区别是，这些并不指向参数绑定的函数指针（虽然很多人惊讶，但 Go 是完全能够实现**柯里化**[1]的），而仅仅是调用时必须传递整数的原始函数。

下面的代码使用了一个新的 Go 关键字 `range`，它可以循环代码段里的内容，并提供一个索引和一个可以调用的函数。

```
var rolls = getDieRolls()
for index, rollFunc := range rolls {
       fmt.Printf("Die Roll Attempt #%d, result: %d\n", index, rollFunc(10))
}
```

现在如果将函数示例当作数据，并重新运行程序，应该会看到如下所示的输出。

```
Die Roll Attempt #0, result: 3
Die Roll Attempt #1, result: 42
```

如果使用之前定义的随机化的掷骰子函数，那么第二个参数会一直打印 **42**。

Go 中的函数与其他编程语言中的类和方法同样重要，不应该错误地认为简单的语法就缺乏能力和实用性。

使用结构体

Go 中的**结构体**只是字段的类型化集合。结构体有很大的灵活性，可以嵌套使用。我们可以创建匿名结构体，也可以创建对结构体进行操作的方法，下一章会介绍。

---

1 网上有很多关于该方面的信息，只要搜索“functional programming golang”即可。

首先，创建一个简单的结构体，其中包含一个人的姓名和年龄。

从 type 关键字开始，先声明一个类型（在使用函数时会看到这个类型），然后是结构体的名称（在这里是 person），最后是关键字 struct。后面的内容是此类型的字段集合。

```
type person struct {
        name string
         age  int
}
```

可以使用不同的方式创建结构体。以下代码都可以用来创建一个 person 结构体。

```
var p =  person{}
var p2 = person{ "bob", 21 }
var p3 = person{ name: "bob", age: 21, }
var p4 = &person{}
var p5 = &person{ "bob", 21 }
var p6 = &person{ name: "bob", age: 21, }
```

第一行展示了如何创建一个空的结构体。第二行通过定义字段的顺序使我们可以顺序地赋初始值。第三行允许以任何我们喜欢的顺序为字段赋值，因为它标明了每个字段的名称。

接下来的三行完成了同样的操作，但不是创建一个结构体，而是创建一个指向结构体的指针。不用担心使用指针——它不像在 C 或 C ++中那么复杂。

我们可以使用传统的点（“.”）语法访问结构变量中的单个字段，这正是大多数人熟悉的方式。

为了扩展这个具有新特性的简单结构体，我们将在一个多人游戏服务器中创建一个多级结构来存储有关潜在敌人的信息，如代码清单 3.3 所示。

代码清单 3.3　go-structs.go

```
package main

import (
        "fmt"
)

type power struct {
        attack  int
        defense int
}
```

```
type location struct {
        x float32
        y float32
        z float32
}

type nonPlayerCharacter struct {
        name   string
        speed  int
        hp     int
        power  power
        loc    location
}

func main() {
        fmt.Println("Structs...")

        demon := nonPlayerCharacter{
                 name:  "Alfred",
                 speed: 21,
                 hp:    1000,
                 power: power{attack: 75, defense: 50},
                 loc:   location{x: 1075.123, y: 521.123, z: 211.231},
        }

        fmt.Println(demon)

        anotherDemon := nonPlayerCharacter{
                name: "Beelzebub",
                speed: 30,
                hp:    5000,
                power: power{attack: 10, defense: 10},
                loc:   location{x: 32.03, y: 72.45, z: 65.231},
        }

     fmt.Println(anotherDemon)
}
```

我们看到的第一个结构体是 power，它包含一组进攻和防守的等级。

接下来可以看到 location 结构体，其中包含一组三维坐标。

最后定义了 nonPlayerCharacter 结构体，其中包含非玩家角色的名称、速度、生命值、位置和力量。理想情况下，这些信息将被在游戏服务器中运行的算法使用，例如确定战斗决议、计算角色之间的距离或冲突以及其他常见的游戏操作。

我们使用结构体创建语法来创建一个新的 NPC（非玩家角色）、demon、第二个

NPC 和 anotherDemon。当打印时，可以看到如下的输出内容。

```
Structs...
Alfred (1075.123047,521.122986,211.231003)
Beelzebub (32.029999,72.449997,65.231003)
```

这里需要指出的是，与原始输入相比，显示的值会有轻微解析率的误差和四舍五入的差异。这很正常，在使用浮点类型的数值时常会出现此类问题。

介绍 Go 接口

在 Go 的所有最受喜爱的功能中，**接口**绝对可以排在前五名。上一节介绍了如何创建包含逻辑分组数据的结构体。下一节将介绍如何扩展结构体的功能以及如何将其应用于接口。

向结构体添加方法

我们可以使用结构体（实际上几乎可以是任何类型）来锚定方法，这意味着可以拥有专门用于结构体的函数。这种特性会使人产生面向对象的错觉，然而这是一个不幸的误解，因为 Go 并不是面向对象的。

下面开始创建一些有趣的结构体。

```
type attacker struct {
        attackpower  int
        dmgbonus     int
}
type sword struct {
        attacker
        twohanded bool
}
type gun struct {
        attacker
        bulletsremaining int
}
```

在上述代码中，我们创建了两种类型：gun 和 sword。这两种类型共享一个名为 attacker 的子结构体。如果名称与数据类型匹配，则不必重复声明两次。不要将此语法与结构继承或任何其他种类的继承混淆。请确保让所有人知道，Go 不是面

向对象的。

现在使用 Go 的语法为结构体 sword 和结构体 gun 添加 Wield 方法，代码如下。

```
func (s sword) Wield() bool {
        fmt.Println("You've wielded a sword!")
        return true
}
func (g gun) Wield() bool {
        fmt.Println("You've wielded a gun!")
        return true
}
```

分别创建 guns、swords 和 wield，代码如下。

```
sword1 := sword{attacker: attacker{attackpower: 1, dmgbonus: 5}, twohanded: true}
gun1 := gun{attacker: attacker{attackpower: 10, dmgbonus: 20}, bulletsremaining: 11}
fmt.Printf("Weapons: sword: %v, gun: %v\n", sword1, gun1)
sword1.Wield()
sword2.Wield()
```

现在可以创建在特定类型上执行操作的方法了，在我们的示例中，这种类型是结构体。为了便于后续引用，本书后面的示例中不会仅将方法锚定到结构体。

Go 中的接口动态类型检查

下面讨论接口以及它们为 Go 语言带来的强大功能。

假设要创建一个自动挥舞武器的函数，无论是枪还是剑。在传统的面向对象语言中，我们可能会创建一个名为 *IWeapon* 或类似的接口，然后显式地声明某些类来实现这个接口。

在 Go 中没有这样明确的要求。可以使用以下的简单代码声明一个指示某物是武器的接口。

```
type weapon interface {
        Wield() bool
}
```

与很多其他语言不同的是，我们不必声明一个结构体或其他类型“是一个武器”。它能凭借 Go 是否可以在一定范围内找到指定签名的方式自动被识别。

很多程序员喜欢将这种情况称之为“鸭子类型”，这源自于“鸭子测试”，大家可以阅读更多来自维基百科的相关引用。

如果它看起来像一只鸭子，像鸭子一样游泳、嘎嘎叫，那么它可能就是一只鸭子。

——鸭子测试

Go真的不关心类型是怎么样，是否会游泳、嘎嘎叫或看上去是否像鸭子。它所要做的就是表现得像只鸭子。因此，我们想进一步提出的理论是，这种类型匹配应该被称为“混合”而不是“鸭子”类型（使用#wibtyping看起来更加紧追潮流）。不论从形状、尺寸、原料或是设计意图来看，它都是混合式的。

有了这个接口，我们就可以创建一个函数，它可以在任何实现挥舞操作的代码中执行挥舞操作。

```
func wielder(w weapon) bool {
    fmt.Println("Wielding...")
    return w.Wield()

}
```

现在可以把sword和gun变量传递给wielder函数。

```
wielder(sword1)
wielder(gun1)
```

输出结果如下。

```
Weapons: sword: {{1 5} true}, gun: {{10 20} 11}
Wielding...
You've wielded a sword!
Wielding...
You've wielded a gun!
```

到现在为止一切正常。但假设有一天我们过得很糟糕，突然想挥舞一把椅子，又将会发生什么？该如何编写代码？

```
chair1 := chair{legcount: 3, leather: true}
wielder(chair1)
```

如果在创建chair结构体之后尝试编译这段代码，将得到一个如下所示的编译输出的错误。

```
cannot use chair1 (type chair) as type weapon in argument to wielder: chair does not implement
weapon (missing Wield method)
```

然而，如果给chair结构体一个`Wield`方法，那么“混合式”（或者这个例子中的“它可以挥舞吗？”）的测试将会通过，得到的输出如下。

```
Weapons: sword: {{1 5} true}, gun: {{10 20} 11}
```

```
Wielding...
You've wielded a sword!
Wielding...
You've wielded a gun!
Wielding...
You've wielded a chair!!You having a bad day?
```

本节中最重要的一点是，不必显式声明类型是什么。如果它满足任何一个接口，那么类型便可以在任何使用该接口的地方使用。在类型作者和接口作者不同时，这将有助于实现极其强大的扩展机制。

回到本章最初创建的一些结构体，我们可以使用方法来增强它们。Go 在 `fmt` 包中包含内置的接口 `Stringer`。符合 `Stringer` 接口的任何内容都可以表示为字符串。在我们的示例中，可以将三维游戏内的位置表示为字符串，代码如下。

```
func (loc location) String() string {
        return fmt.Sprintf("(%f,%f,%f)", loc.x, loc.y, loc.z)
}
```

记住，我们并没有明确告知 Go 该结构体“实现了 Stringer”，我们通过暴露一个 `String` 方法隐式地满足了协议要求。还可以添加更多的实用方法，例如计算位置之间的距离（从而允许计算两个 NPC 之间的距离），代码如下。

```
func (loc location) euclideanDistance(target location) float64 {
        return math.Sqrt(
                (loc.x-target.x)*(loc.x-target.x) +
                        (loc.y-target.y)*(loc.y-target.y) +
                        (loc.z-target.z)*(loc.z-target.z))
}

func (npc nonPlayerCharacter) distanceTo(target nonPlayerCharacter) float64 {
        return npc.loc.euclideanDistance(target.loc)
}
```

上述代码可以实现在游戏中调用这些工具方法以进行调试和计算。

```
fmt.Printf("Npc %v is %f units away from Npc %v\n", demon, demon.distanceTo(anotherDemon),
anotherDemon)
```

我们甚至可以创建一个名为 `Distancer` 的接口，允许函数对一整套类型进行操作，这些类型能够计算它们与其他点之间的距离。

使用第三方包

Go 语言鼓励开发人员通过使用包创建小型、可复用的软件组件。本章中一直在

使用软件包，尽管迄今为止还没有实现什么重要的功能。

例如，我们使用 fmt 包向控制台打印和格式化字符串，使用 math 包获取计算欧氏距离所需的平方根函数，使用 math/rand 包获取随机数。

到目前为止，我们一直在使用核心 Go 库中的包。因此所需的包都已经包含在 Go 工作区（$GOPATH 下的文件夹树）中了。

然而这只是冰山一角。很多有才华和热情的开发人员编写了大量公开发布的软件包，包括从构建 Web 程序、基于内存的图表到打印 ASCII 艺术横幅的所有库。

下面的示例将使用名为 tablewriter 的包。这个包托管在 GitHub 中，它允许我们在控制台上打印花哨、正确格式的表格。这个功能看起来可能微不足道，但大家一定会惊讶于使用该包需要在控制台和日志中打印列式数据的次数。

在使用这个包之前，需要先获取它。在 Go 中可以使用命令提示符执行 go get 命令，如下所示。

```
go get github.com/olekukonko/tablewriter
```

该命令会首先询问是否下载包，并将其放在 Go 工作区中。如果正确执行这条命令，应该看不到任何输出，但可以看到目录 $GOPATH/src/github.com/olekukonko/tablewriter 下会出现* .go 文件、一些 Markdown（稍后讨论）文件和一个子目录。简而言之，这都是 GitHub 仓库包含的内容。

大多数软件包开发人员会在他们的 GitHub 仓库中提供一个 README 文件，解释如何使用他们开发的软件包。在继续介绍代码示例之前，请查看 https://github.com/olekukonko/tablewriter 中 tablewriter 的 README 文件。

现在将开始使用第三方库，我们已经通过 go get 获取了它们。首先创建一个 Go 应用程序，它使用 table writer 输出了一个在本章前面示例中出现的 NPC 列表。此包的使用示例见代码清单 3.4。

代码清单 3.4 go-package-consumer.go

```
package main

import (
        "os"

        "github.com/olekukonko/tablewriter"

)
```

```
func main() {
        data := [][]string{
                []string{"Alfred", "15", "10/20", "(10.32, 56.21, 30.25)"},
                []string{"Beelzebub", "30", "30/50", "(1,1,1)"},
                []string{"Hortense", "21", "80/80", "(1,1,1)"},
                []string{"Pokey", "8", "30/40", "(1,1,1)"}}
        }
        table := tablewriter.NewWriter(os.Stdout)
        table.SetHeader([]string{"NPC", "Speed", "Power", "Location"})
        table.AppendBulk(data)
        table.Render()
}
```

以上代码中的大部分内容是从作者的示例中复制的，我们已经更改了列标题和数据来表示访问虚拟游戏服务器的玩家和他们的位置。

运行应用程序可以得到以下输出。

```
$ go run go-package-consumer.go
+-----------+-------+-------+-----------------------+
|    NPC    | SPEED | POWER |       LOCATION        |
+-----------+-------+-------+-----------------------+
| Alfred    |    15 | 10/20 | (10.32, 56.21, 30.25) |
| Beelzebub |    30 | 30/50 | (1,1,1)               |
| Hortense  |    21 | 80/80 | (1,1,1)               |
| Pokey     |     8 | 30/40 | (1,1,1)               |
+-----------+-------+-------+-----------------------+
```

如果无法通过编译，或者出现与 tablewriter 相关的导入失败，请确保已经正确下载软件包，并且是在 Go 工作区中构建此示例代码的。

我们不赞成重新发明“轮子”，因此可以在 Go 的大量第三方库中使用各式各样的“轮子”。这意味着，在可预计的未来，将会有更多各种各样用于 Go 的“轮子”。

创建自有包

我们已经了解了如何使用结构体、创建简单函数、创建结构体方法以及如何使用其他人构建的包。下面就可以开始创建自己的包了。

Go 的公共库、开源包也许足以解决所有的问题，我们可能永远也不需要创建一个自有包。然而，如果想要解决一个新的问题或编写一套优雅的代码集，我们希望分享，也应该分享它。Go 中所有的代码都是开源的，正如我们每次创建一个可复用

的包，都会将它放到GitHub上以便其他人使用一样。

在Go中创建一个包实际上很简单。到目前为止，所有代码都放在main包中。Go会进入main方法，然后代码将被当作一个应用程序执行。非main包需要被其他的Go代码导入使用。

导出函数和数据

如果一直密切关注目前为止所有的代码示例，相信大家可能已经注意到，所有的结构体、字段和函数都是以小写字母开头的，所有我们使用的其他包的函数都是以大写字母开头的。这不是一种和其他语言一样的随意的命名规则，这种命名规则对编译结果具有实际的影响。

软件包是Go中的一个访问范围单位。我们没有OOP中的成员访问关键字，如public或private，但仍然可以控制代码的可见性。我们在Go中创建的任何以小写字母开头的类型（包括函数）都被认为是包所私有的或者没有导出的。我们创建的任何以大写字母开头的类型都会被导出，任何使用该包的人都可以看到。

作为一个软件包开发人员，可以这样认为：使用大写字母的代码都是公共API，而使用小写字母的则仅供内部使用。

创建包

不与其他人分享我们以非玩家角色所做的有意思的工作，将是一件令人惭愧的事，所以首先创建一个包，将开创性的代码开源。

先在go-primer目录下创建一个npcs目录，这意味着我们将在$GOPATH/src/github.com/cloudnativego/go-primer/npcs目录中进行工作。也可以在其他目录中执行此操作，但如果不是位于有效的Go工作区中，情况将会变得非常糟糕。

按照惯例，许多软件包开发人员更喜欢创建一个types.go文件，其中包含要由软件包使用或导出的类型。一些开发人员喜欢将导出类型和私有类型拆分为两个不同的文件，但这通常只在存在很多类型时使用。与本书中展示的所有模式一样，大家可以任意使用或拒绝使用，但希望在整体示例中保持一致。

代码清单3.5列出了types.go文件中包含的内容。

代码清单3.5 npcs/types.go

```
package npcs
```

```
// Power describes the attack and defense power of an NPC
type Power struct {
        Attack  int
        Defense int
}

// Location describes where in the virtual world an NPC exists
type Location struct {
        X float64
        Y float64
        Z float64
}

// NonPlayerCharacter represents metadata for an in-game creature
type NonPlayerCharacter struct {
     Name     string
     Speed    int
     HP       int
     Power    Power
     Loc      Location
}
```

上述代码可能看起来不同于之前实现的 NPC 结构体。现在所有结构体都是以大写字母开头的，就像它们中的所有字段一样。如果大家喜欢，也可以在一个结构体中混合地声明导出字段和私有字段。此外，我们为每种导出类型都添加了注释。这不仅是一个好的习惯，如果要用到文档生成工具，那么这更是大部分 Go 静态分析器强制执行的规则。

所有向包使用者暴露的方法都会放在一个名为 `npcs.go` 的单独文件中，如代码清单 3.6 所示。

代码清单 3.6 `npcs/npcs.go`

```
package npcs

import (
        "fmt"
        "math"
)

func (loc Location) String() string {
        return fmt.Sprintf("(%f,%f,%f)", loc.X, loc.Y, loc.Z)
}
```

```
// EuclideanDistance returns the distance between two in-game locations
func (loc Location) EuclideanDistance(target Location) float64 {
        return math.Sqrt(
                (loc.X-target.X)*(loc.X-target.X) +
                        (loc.Y-target.Y)*(loc.Y-target.Y) +
                        (loc.Z-target.Z)*(loc.Z-target.Z))
}

// DistanceTo returns the distance between two in-game characters
func (npc NonPlayerCharacter) DistanceTo(target NonPlayerCharacter) float64 {
        return npc.Loc.EuclideanDistance(target.Loc)
}

func (npc NonPlayerCharacter) String() string {
        return fmt.Sprintf("%s  %s", npc.Name, npc.Loc)
}
```

导出以上所有函数，都需要以大写字母开头，同时它们操作的结构体和字段也已经被导出。

创建这个子包后，便可以在客户端应用程序中使用它了，如代码清单 3.7 所示。

代码清单 3.7　custom-package-consumer.go

```
package main

import (
        "fmt"

        "github.com/cloudnativego/go-primer/npcs"
)

func main() {
        mob := npcs.NonPlayerCharacter{Name: "Alfred"}
        fmt.Println(mob)

        hortense := npcs.NonPlayerCharacter{Name: "Hortense",
              Loc: npcs.Location{X: 10.0, Y: 10.0, Z: 10.0}}
        fmt.Println(hortense)

        fmt.Printf("Alfred is %f units from Hortense.\n",
              mob.DistanceTo(hortense))
}
```

现在，我们已经通过更好的方式构建了庞大的 NPC 包，下面可以开始运行示例包，并享受它带来的成就感。

```
$ go run custom-package-consumer.go
Alfred    (0.000000,0.000000,0.000000)
Hortense (10.000000,10.000000,10.000000)
Alfred is 17.320508 units from Hortense.
```

本章小结

本章简要介绍了 Go 语言的基础知识。这一章不是一个完整的语言教程，甚至算不上是语言参考。本书的范围只局限在介绍微服务和云模式的实践上，但同时，我们也希望大家在接触构建微服务的细节之前，已经了解了一些 Go 语言的基础知识。

如果觉得这一章有点难以理解，那么可以先停下，重新阅读我们在本章开头时提到的那几个章节。已经具有 Go 开发经验的读者可以通过本章巩固知识点。

4

持续交付

编程不是一种零和游戏。将知识传授给程序员同行并不会减少自身的知识储备。

——John Carmack

编写软件如同一种需要很多年才能完善的工艺。我们在职业生涯早期经常会将代码的行数与应用程序的质量或闪亮的功能点等同：编写的代码行数越多，我们就是越优秀的开发人员。

一旦消除了这种误解，我们就会走向另一条路。坚信编写的代码行数越少，代码质量就越高，我们就越是一名优秀的开发人员。

在开发技能发展到某种程度时，我们开始意识到代码行数与应用程序质量之间并没有直接关系。无论追求高或低的技术水准，在现实中，代码行数总是会令人感到困扰。

我们需要从极度糟糕和错误的事情中汲取宝贵经验。深夜里跟所有 IT 人员一起在电话中讨论，全力以赴地面对一个在劫难逃的、被称之为“发布”的事件所带来的影响时，观点也会随着一些事情而改变。

最终我们得出一致结论：实际上我们所追寻和期望得到的东西是信心。我们希望被告知，在应用程序部署之前，它便可以按照既定的方式运行。不能通过取消令人恐惧的发布而树立信心，而是应该通过时刻发布来树立信心。

本章将讨论持续集成和持续交付，包括以下几个话题。

- Docker 和 Docker Hub，基于云 CI 和不可变部件的强大工具。
- Wercker，一种适用于独立开发者、初创公司甚至是完整企业的 CI 工具。

- 在每次提交 Git 后自动构建项目。
- 将构建工件作为 CI 流水线的一部分自动部署。

Docker 介绍

Docker 是一种使用 Linux 内核功能的容器工具，如 **cgroups** 和**命名空间**，它能提供网络、文件和内存资源的隔离，而无须依靠一个完整的虚拟机。最近，Docker 一直保持巨大的发展势头，并且应用在越来越多的行业中。

为什么要使用 Docker

Docker 能够帮助我们创建一个固定的软件版本，它可以在任何地方运行，无论目的环境中有没有安装该软件（但是必须安装 Docker）。例如，如果向 Docker Hub（http://hub.docker.com）部署一个包含服务的 Docker 镜像，那么在机器上安装了 Docker 的任何人都可以运行该服务，不必担心依赖项的安装，不必考虑编译器或任何其他需要支持的基础设施，开发机器也不会因为安装了这个服务的配置和依赖项而受影响。所有内容都包含在 Docker 镜像中。

有关 Docker 的更多信息，包括如何创建自有的 Docker 文件和镜像以及有关高级管理的详细信息，请查阅 Karl Matthias 和 Sean P. Kane 的书籍 *Docker Up and Running*. O'Reilly Media。

正如本章后面即将提到的，我们可以直接使用持续集成工具向 Docker Hub 发布 Docker 镜像——所有这些都是在云端完成的，我们几乎没有在本地工作站上安装任何基础设施。

安装 Docker

在 Mac 上安装 Docker，首选的方法是使用 Docker *Toolbox*。在一些较旧的文档中可能会提及一个名为 *Boot2Docker* 的工具，它已经被弃用，不应该通过这种方式安装 Docker。

有关在 Mac 上安装 Docker 的详细信息，请查看链接 https://docs.docker.com/engine/installation/mac/。在写这篇文章的时候，我们已经安装了 Docker 1.8.1 版本。请确保在执行安装操作之前已经查看了最新的安装说明。

如果恰巧有一台 2010 年以后生产的具有 4GB 内存的 Mac，且没有安装 VirtualBox，同时操作系统为 Yosemite 或更新的版本，那么只需安装原生的 Docker 程序，而无须安装 VirtualBox 虚拟化。具体请查看 Docker 网站的说明。

我们编写了如何使用 Docker Toolbox 的示例，如果正在使用新 Mac 的原生应用程序，那么这个过程可能略有不同（当然希望更顺利）。

> **通过 Homebrew 安装**
>
> 可以通过 Homebrew 手动安装 Docker 并完成所有预配置项。这样虽然会带来一些额外工作，但它易于维持版本更新且与已安装的其他软件一致。如果不打算使用 Docker Toolbox，则需要使用 Homebrew 来安装 Docker、Docker Machine 和 VirtualBox。

如果已经正确安装了 Docker Machine，那么我们要做的第一件事就是启动它。由于 Docker 依赖于特定的 Linux 内核功能，所以实际上启动的是一台虚拟机（由 VirtualBox 提供支持），该虚拟机模拟 Linux 内核功能来启动一个 Docker 服务器（称为 *daemon*）。

要启动 Docker Machine，请在终端上输入以下命令。

```
$ docker-machine start default
Starting VM...
Started machines may have new IP addresses. You may need to re-run the 'docker-machine
env' command.
```

如果已经修改了机器的默认名称，那么请将 `default` 修改为机器的名称。如果在 Docker Toolbox 安装期间选择默认配置，那么它将安装一台名为 `default` 的 Docker 机器。

根据计算机的电池属性，可能需要花费几分钟的时间来启动 Docker。这时需要配置一些环境变量（在每一个打算使用 Docker 的终端窗口中），以便与 Docker 服务器进行正常通信，这些命令包括启动应用程序、列出仓库里的镜像等。可以通过以下命令在终端打印出需要注意的配置。

```
$ docker-machine env default
export DOCKER_TLS_VERIFY="1"
export DOCKER_HOST="tcp://192.168.99.100:2376"
export DOCKER_CERT_PATH="/Users/khoffman/.docker/machine/machines/default"
export DOCKER_MACHINE_NAME="default"
# Run this command to configure your shell:
```

```
# eval "$(docker-machine env default)"
```

以下是配置正确的 Docker 环境所需的 shell 命令。要使这些配置在启动 Docker 时生效，可以使用以下命令。

```
$ eval "$(docker-machine env default)"
```

Docker Toolbox 提供了一种预配置 Docker shell 的快捷方式，允许跳过这一步，但是我们发现它很臃肿和复杂，因此只需运行 `eval` 命令就可以使一个已有的终端窗口 Docker 化。

现在应该能在终端上运行所有 Docker 命令来检查安装了。可以发现平时最常用的命令是 `docker images`，这个命令会打印出存储在本地仓库中的所有 Docker 镜像。

运行 Docker 镜像

这是最有趣的部分！可以手动下载和安装 Docker 镜像，让它们在计算机上可用，或者也可以直接从 Docker Hub（或在企业环境中配置的私有镜像仓库）上运行它们。

例如，执行以下命令将直接从 Docker Hub 上启动“hello world”Web 服务器示例。

```
$ docker run -p 8080:8080 cloudnativego/book-hello
[negroni] listening on :8080
[negroni] Started GET /
[negroni] Completed 200 OK in 71.688µs
```

以上输出显示了镜像如何在本地缓存。如果是第一次执行，那么将看到许多进度报告，表明正在下载 Docker 镜像。

此命令将 Docker 镜像内部的 8080 端口映射到外部的 8080 端口上。如上所述，Docker 提供了网络隔离，所以除非明确定义了来自外部的流量在容器内的路由规则，否则隔离本质上将是一个防火墙。

由于我们已经定义了内部和外部的端口映射关系，因此可以通过 Docker 机器的 IP 地址加上 8080 端口进行访问（注意不是 `localhost`）。

> **注意**
>
> 如果机器正在运行，则可以使用 `docker-machine ip default` 命令在 Docker 上查看该 Docker 机器的 IP 地址。如果机器的名称不是 `default`，替换成真实机器名称即可。

```
$ curl http://192.168.99.100:8080
Hello from Go!
```

此命令显示了从 Docker Hub 下载一个功能完整的软件，在本地缓存镜像并使用 docker run 执行 Docker 镜像的过程。即使没有安装 Go 工具或配置 Go 工作区，我们仍然可以使用这个 Docker 镜像启动示例服务。

一旦开始使用 Docker 提供的所有功能和工具，我们就会感受到未来巨大的可能性，例如将构建工件封装成 Docker Hub 镜像。

> **提示**
>
> 如果使用了很多不同的 Docker 镜像进行大量开发，存储这些镜像可能会占满虚拟机磁盘。这些镜像都只是高速缓存，所以偶尔清理并不会造成永久性的后果。如果遇到这种情况，以下两个 shell 命令会非常有帮助。
>
> 删除所有 Docker 容器：`docker rm $(docker ps -a -q)`
>
> 删除所有 Docker 镜像：`docker rmi $(docker images -q)`

与 Wercker 的持续集成

大家过去可能拥有持续集成服务的经验。有一些软件非常受欢迎，包括 Jenkins、TeamCity 和 Concourse 等，在微软中则是 Team Foundation Server（TFS）。在本章以及本书的其余部分中，我们将使用名为 **Wercker** 的 CI 工具。

以上工具都尝试通过软件方式帮助开发人员完成 CI 最佳实践。在这一节中，我们将提供关于 CI 的简要概述，然后开始配置 Wercker 来自动构建应用程序。

持续集成的最佳实践

在浏览维基百科时，我们偶尔会在无尽的琐碎中发现智慧的珍珠。维基百科中关于持续集成（CI）最佳实践这部分的内容特别有用且丰富。以下是对相关内容的说明，包括这些准则如何适用于本书所做的工作中。

- **维护代码仓库**：正如第 2 章中所讨论的，熟悉 GitHub 对于完成本书中的工作是必不可少的。
- **自动化构建**：配置自动化构建是本章的目标。
- **使构建自测试**：测试驱动开发不应只是偶尔采用的方式，它是云之道的一部

分，具体见第 5 章。我们编写的所有代码都要经过测试，CI 服务器必须可以执行这些测试。

- **每天向主库提交代码**：云之道的另一个准则是尽早发布，时刻发布。通过不断向主库提交代码来体现 CI 的最佳实践。等待代码提交到主库（分支是用于生产版本的发布）之前的时间越长，信心指数就越低，就会越恐惧发布。
- **每个提交都执行构建**：在每次提交代码时，CI 工具都应执行构建，其中包括运行单元测试、评估测试覆盖率以及经常运行的静态分析工具。
- **保持构建速度**：如果构建流程太慢或自动化流程过于烦琐，便意味着某些环节可能已经出错。构建应该很快，我们应该可以看到它快速通过或失败。缓慢、臃肿、长时间的构建过程最终会变得令人厌恶，并让人产生想要将它们关闭或停止测试的想法。
- **在生产的克隆环境中测试**：本书将展示实现这一目标的多种方法，包括将服务部署到本地云上。
- **易于获得交付成果**：正如我们在上一节中提到的，可交付成果将放在 Docker Hub 上，这使得获取构建工件变得异常容易。然而，Wercker 不强迫我们使用 Docker Hub，因此可以从网站或命令行工具中获取构建工件。
- **每个人都可以看到最新的构建结果**：如果没人知道构建失败，那么就绝对没有理由进行自动构建。如果一个构建变成红色状态，那么使它通过就应该立即成为首要的任务。在 Wercker 上很容易看到构建的结果，也有一个客户端应用程序可以在构建失败时立即通知我们。重要的是，工具会使构建状态可见，但是要由个人或团队依据严重程度来制定处理失败构建的规则。
- **自动部署**：我们将在本章中看到如何使用 Wercker 在成功构建结束时自动部署 Docker 镜像。如果部署是手动的过程，那么将会非常容易出错。手动部署是缓慢和烦琐的，团队应尽量避免使用手动方式，因为它会在部署中产生更长的延迟，使我们对系统丧失信心。

为什么使用 Wercker

所有的 CI 工具都可以在市面上获取，但为何要建议使用 Wercker 呢？依据云之道的准则评估了所有工具，发现 Wercker 正是我们需要的。

首先，无须在工作站中安装 Wecker，仅安装一个命令行客户端即可，构建过程

全部在云端进行。

其次，不用通过信用卡就可使用 Wercker。当我们迫切希望简化流程时，这是一件令人赞叹的事。付款承诺这一条件大大增加了开发者的压力，这通常是不必要的。

最后，Wercker 使用起来非常简单。它非常容易配置，不需要经过高级培训或拥有持续集成的博士学位，也不用制定专门的流程。

> **提示**
>
> 如果在构建工具上花费的调试时间比开发时间还多，那么可以肯定是使用了错误的工具。

通过 Wercker 搭建 CI 环境只需经过三个基本步骤。

1. 在 Wercker 网站中创建一个应用程序。
2. 将 `wercker.yml` 添加到应用程序的代码库中。
3. 选择打包和部署构建的位置。

创建 Wercker 应用程序

通过 wercker.com 创建一个账户。可以选择使用 GitHub 账户登录到 Wercker 上，这样简化了所有流程，并降低了验证凭证的风险。这也使得 Wercker 在配置构建流程时更容易访问 GitHub。

一旦拥有了账户，那么只需简单地点击位于顶部的**应用程序**菜单，然后选择**创建**选项即可。如果系统提示是否要创建组织或应用程序，请选择创建**应用程序**。Wercker 组织允许多个 Wercker 用户之间进行协作，而无须提供信用卡。图 4.1 为设置新应用程序的向导页面。Wercker 总是在更改和升级 UX 和功能，所以当阅读到此处时，截图所示内容可能已经过时。

不必具体遵循以上内容，因为稍后会在创建 Wercker 构建这一章中进行练习。在选择 GitHub 或 Bitbucket 之后，系统会提示我们选择仓库。这是 Wercker 将要链接的新应用程序的源码库。图 4.2 显示了 Wercker 中应用程序的构建列表。

当在此仓库上进行提交时，Wercker 将根据我们的首选项和配置启动构建流程。

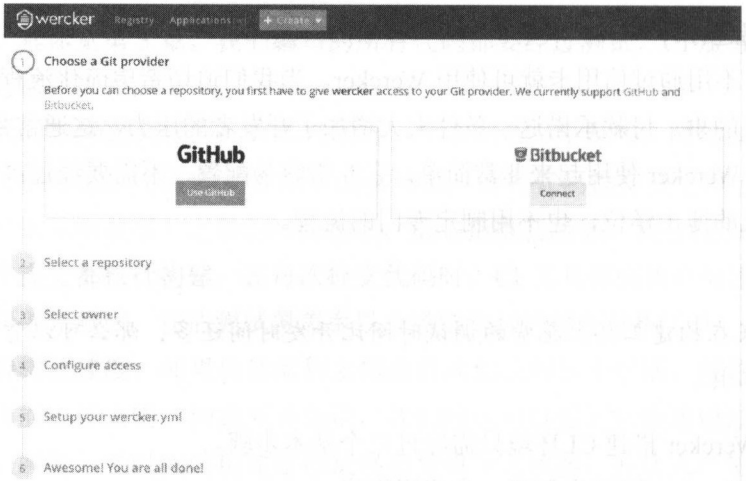

图 4.1　在 Wercker 中创建应用程序
来源：wercker.com

图 4.2　Wercker 中的应用程序列表
来源：wercker.com

在创建应用程序的第五步，Wercker 会尝试生成一个 `wercker.yml` 文件（后面会讨论）。不过至少对于 Go 应用程序来说，这个配置很少会满足要求，所以我们总是需要创建自己的 Wercker 配置文件。

安装 Wercker CLI

我们总是希望能在本地进行 Wercker 构建，以便可以了解在云端构建的过程如何进行。本地构建和云端构建都依赖于 Docker 的使用。基本上，代码会被置于所选择

的 Docker 镜像中（在 `wercker.yml` 中定义），然后再选择执行的内容和方法。

要在本地运行 Wercker 构建，需要使用 Wercker CLI。有关如何安装和测试 CLI 的内容，请查看 http://devcenter.wercker.com/learn/basics/the-wercker-cli.html。Wercker 更新文档的频率要比本书更高，所以请在本书中做个标记，然后根据 Wercker 网站的文档安装 Wecker CLI。在撰写本书时，Wercker 文档中仍然是使用 Boot2Docker 安装 Docker 的。可以忽略这一点，并使用之前提供的方法来安装和运行 Docker，相关内容可以在“获取 CLI”部分进行查阅。

如果已经正确安装了 CLI，应该可以查询到 CLI 的版本，代码如下所示。

```
$ wercker version
Version:  1.0.295
Compiled at: 2015-10-23 06:19:25 -0400 EDT
Git commit: db49e30f0968ff400269a5b92f8b36004e3501f1
No new version available
```

如果正在使用 CLI 的旧版本，则会看到以下的自动更新提示。

```
$ wercker version
Version:  1.0.174
Compiled at: 2015-06-24 10:02:21 -0400 EDT
Git commit: ac873bc1c5a8780889fd1454940a0037aec03e2b
A new version is available: 1.0.295 (Compiled at: 2015-10-23T10:19:25Z, Git commit:
db49e30f0968ff400269a5b92f8b36004e3501f1)
Download it from: https://s3.amazonaws.com/downloads.wercker.com/cli/stable/
darwin_amd64/wercker
Would you like update? [yN]
```

如果在选择自动更新后出现问题（我们曾遇到过几次），使用 Wercker 文档中提到的简单方法，重新运行 `curl` 下载最新的 CLI 即可。

创建 Wercker 配置文件

Wercker 配置文件是一个 YAML 文件，主要包含三个部分，对应可用的三个主要流水线。

- **Dev**：定义了开发流水线的步骤列表。与所有流水线一样，可以选定一个 **box** 用于构建，也可以全局指定一个 box 应用于所有流水线。box 可以是 Wercker 内置的预制 Docker 镜像之一，也可以是 Docker Hub 托管的任何 Docker 镜像。
- **Build**：定义了在 Wercker 构建期间要执行的步骤和脚本的列表。与许多其他服务（如 Jenkins 和 TeamCity）不同，构建步骤位于代码库的配置文件中，而

不是隐藏在服务配置里。

- **Deploy**：在这里可以定义构建的部署方式和位置。

> YAML
>
> 根据 http://www.yaml.org/ 中的介绍，YAML 代表“YAML Ain't Markup language”，是一种人类可读、机器可解析的文本格式。也许很多人会不同意这个观点，但我们认为，YAML 之于 JSON，类似 JSON 之于 XML。不建议使用 YAML 作为 RESTful 服务的有效内容格式，但它却非常适用于配置和存储元数据。

在编写本书时，Wercker 提出了**工作流**的概念，通过使用分支、条件构建、多个部署目标和其他高级功能扩展了流水线的功能。为了简单起见，本书中将重点关注流水线。

编辑 YAML 配置文件时应该谨慎。虽然在传统意义上，我们习惯使用 IDE 的标准文本编辑器来编辑文件，但实际上可能更希望使用能进行正确解析和格式化的 YAML 编辑器。YAML 对空格和制表符非常敏感，如果关闭了间距，有些内容将会损坏。大多数编辑器，包括 Emacs 和 Atom，都有可以处理 YAML 的插件。

如果想了解更多，请查看 http://www.yaml.org/start.html。要阅读代码清单 4.1 中的 YAML，只需了解以下几个 YAML 的主要特点。

- YAML 发音为“yeah-mul”。
- 通过一个或多个空格的缩进表示结构。
- 序列中的元素前添加“-”。
- 键值对（如映射项）的键值之间由没有空格的冒号进行分隔。

最后，如果无法确定 YAML 的格式，又想要进行语法验证，可以访问一个在线 YAML 验证程序，地址是 http://www.yamllint.com/。这个工具已经拯救了很多人。

代码清单 4.1 为 *hello world* 示例中的 `wercker.yml` 文件。

代码清单 4.1　wercker.yml

```
box: golang

dev:
  steps:
    - setup-go-workspace:
        package-dir:  github.com/cloudnativego/hello
```

```
  - script:
      name: env
      code: env

  - script:
      name: go get
      code: |
        cd $WERCKER_SOURCE_DIR
        go version
        go get -u github.com/Masterminds/glide
        export PATH=$WERCKER_SOURCE_DIR/bin:$PATH
        glide install

  - internal/watch:
      code: go run main.go
      reload: true

build:
  steps:
  - setup-go-workspace:
      package-dir: github.com/cloudnativego/hello

  - script:
      name: env
      code: env

  - script:
      name: go get
      code: |

        cd $WERCKER_SOURCE_DIR
        go version
        go get -u github.com/Masterminds/glide
        export PATH=$WERCKER_SOURCE_DIR/bin:$PATH
        glide install

  # Build the project
  - script:
      name: go build
      code: |
        go build

  # Test the project
  - script:
      name: go test
      code: |
        go test -v $(glide novendor)
```

```
    - script:
        name: copy files to wercker output
        code: |
          cp -R ./ ${WERCKER_OUTPUT_DIR}

deploy:
 steps:
    - internal/docker-push:
        username: $DOCKER_USERNAME
        password: $DOCKER_PASSWORD
        cmd: /pipeline/source/hello
        port: "8080"
        tag: latest
        repository: cloudnativego/book-hello
        registry: https://registry.hub.docker.com

    - cng/cf-deploy:
        api: $API
        user: $USER
        password: $PASSWORD
        org: $ORG
        space: $SPACE
        appname: wercker-step-hello
        docker_image: cloudnativego/book-hello
```

此文件包含三个流水线：dev、build 和 deploy。在开发流程中，我们使用 Wercker 和 Docker 创建一个干净的 Docker 镜像，然后直接运行应用程序。

在代码清单 4.1 中，box 键的值是 golang。这意味着我们使用的是一个基础的 Docker 镜像，它已经安装了 Go 环境。这一点至关重要，因为执行 Wercker 构建的基准 Docker 镜像需要包含应用程序所需的构建工具。

这部分中存在一些难以理解的概念。当使用 Wercker 进行构建时，其实并没有使用本地工作站的资源（即使在技术层面上，构建也是在本地执行的），相反，使用的是 Docker 镜像中的可用资源。因此，如果要使用 Wercker 编译 Go 应用程序，需要首先运行包含 Go 的 Docker 镜像。如果想要构建唯一的工件，无论它是在本地还是在 Wercker 的云端运行，使用 Docker 镜像都是完全合理的。

本次构建中运行的第一个脚本是 go get。无论为脚本设置什么名称，构建输出都会有所显示，如图 4.3 所示。

执行 go get 可以获取可能需要的、但不包含在基础镜像中的任何东西。在示

例中，我们使用一个名为 Glide 的工具来帮助管理和获取 Go 依赖项。 Glide 位于 Go 1.5 及更高版本中新增的依赖关系管理功能之上，帮助我们更新依赖关系并固定在某个版本上，依赖现有的 Go 工具则很难实现。

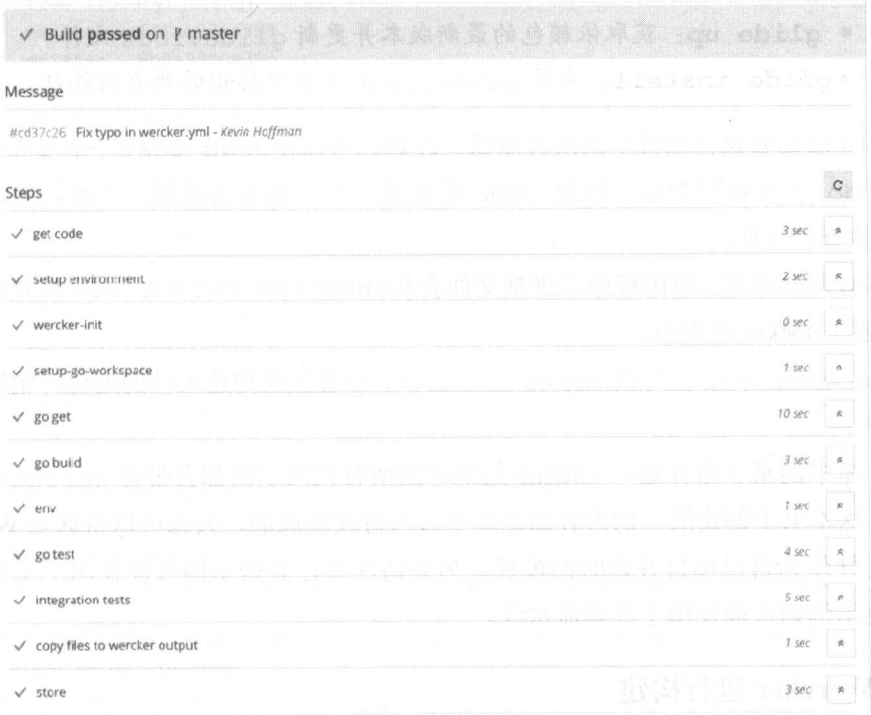

图 4.3 构建步骤执行状态

来源：wercker.com

下一个脚本 `go build` 执行了实际的构建流程。由于我们已经安装了 Glide 并使用 Go 1.6 版本（或更高版本），因此可以使用简单的 `go build` 命令进行编译。

该命令将在 Go 的临时工作区 `vendor/` 中执行，该工作区在 Glide 初始化并签入源代码控制器时创建。这个目录和 Glide 工具会提供所需的任何依赖，并和应用程序一并存储。这样虽然会使 GitHub 仓库有些臃肿，但当涉及构建流程的确定性时，存储的每个字节都是值得的。

如果没有像 Glide 之类的依赖管理工具，我们可能会在构建或部署过程中随时被安装依赖的请求打断。这完全违背了推送不可变工件的目标。

使用 Glide

我们需要的所有关于 Glide 的文档都可以在 http://github.com/Masterminds/glide 中获取。日常开发过程中会用到三个命令。

- **glide init**：基于项目依赖关系创建 glide.yml 和 glide.lock 文件。
- **glide up**：获取依赖包的最新版本并更新 glide.lock 文件。
- **glide install**：安装 glide.lock 文件中标识的所有软件包。

构建后需要执行项目中的所有测试。注意，可以使用 go test -v $(glide novendor) 来执行测试。根据 Glide 的配置，以上命令会返回一个值，告诉我们 Go 要检查的文件。

假设测试通过，应用程序二进制文件会从$WERCKER_OUTPUT_DIR 被复制到最终生成的 Docker 镜像中。

wercker.yml 文件的 deploy 部分显示了部署生成构建工件的过程，稍后将详细讨论。

从本书的第 5 章开始， GitHub 组织中的所有代码示例都会带有 wercker.yml 文件。这本书中创建的一切内容都是基于云与持续集成的，大家可以将这些 Wercker 配置文件作为自己项目开始时的模板。另外请注意，有些示例可能使用了工作流，而其他示例则可能使用了经典流水线。

使用 Wercker 进行构建

触发 Wercker 构建最简单的方法是提交代码。一旦配置了 Wercker，构建便会在代码推送后的几秒钟内开始。当然，我们仍然希望使用常规的 Go 命令行（例如 go build、go test 等）在本地构建和测试应用程序。接下来的步骤可以查看应用程序如何使用 Wercker 流水线（在一个独立、可移植的 Docker 镜像中）进行构建。这对于解答在开发项目期间经常出现的“在我的机器上是可以正常运行的”这一问题是很有意义的。

我们通常在脚本中使用如下的 wercker build 命令。

```
rm -rf _builds _steps _projects
wercker build --git-domain github.com --git-owner cloudnativego --git-repository
gogo-service
rm -rf _builds _steps _projects
```

运行以上脚本（假设存在与 github.com/cloudnativego/gogo-service

仓库相关联的 Wercker 构建），会输出以下内容。

```
$ ./buildlocal
--> Executing pipeline
--> Running step: setup environment
Pulling  from library/golang: latest
Already  exists:  6d1ae97ee388
Already  exists:  8b9a99209d5c
Already  exists:  2e05a52ffd47
Already  exists:  80887d145531
Already  exists:  ec064956c4f0
Already  exists:  c8a688c71293
Already  exists:  03f519453f95
Already  exists:  b449de9eb16c
Already  exists:  7ab9945f3cbe
Already  exists:  7cb2cf0c147e
Already  exists:  b56020e12f7a
Already  exists:  1c3154f0cc14
Already  exists:  d556268f76ae
Already  exists:  0eca3aede538
Digest:  sha256:1cbd73c3a58097c777d85ad289aed4d0af45325288899ddae5a082d5e7a49c46
Status:  Image is up to date for  golang:latest
--> Running step: wercker-init
--> Running step: setup-go-workspace
Using /go as root dir
package-dir option not set, will use default: /go/src/github.com/cloudnativego/
gogo-service
$WERCKER_SOURCE_DIR now points to: /go/src/github.com/cloudnativego/gogo-service
Go workspace setup finished
--> Running step: go get
go version go1.5.2 linux/amd64
--> Running step: go build
--> Running step: go test
Ok  github.com/cloudnativego/gogo-service  0.005s
--> Running step: copy files to wercker output
--> Steps passed: 17.92s
--> Pipeline finished: 19.03s
```

类似“Status: Image is up to date”这样的输出都是命令行工具在确认我们是否已经使用了最新的基准镜像（`library/golang: latest`，如输出中所示）。

注意

请确认基准镜像的版本，或选择特定的发行版本。若 Go 的版本在最新镜像中有所更改，我们可能会遇到编译错误的问题。如果最新镜像中 Go 的版本与在 Wercker 以外使用的版本不同，则可能会导致构建不一致或失败。

其余的输出部分显示了 wercker.yml 中定义的步骤，如 go get、go build 和 go test。我们可以在执行 go test 时添加参数-v，从而获取执行的每一个测试和结果。

部署到 Docker Hub

一旦可以使用 Wercker CLI 在本地构建应用程序，以及可以通过 Git 提交或在网站上点击按钮的方式来触发构建，我们就可以开始安排部署了。

默认情况下，每个构建工件仅在 Wercker 中可用。可以通过网站或者使用 Wercker 命令行进行下载。

另外，还可以把生成的工件部署到多个位置上。Wercker 包含几个内置的部署步骤，也支持很多部署到其他对象的插件（本书后面的章节中将会提到，我们编写一个 Wercker 步骤，并将 Docker Hub 镜像部署到 Cloud Foundry 上）。

首先，可以在 Wercker 网站上设置一个部署对象，如图 4.4 所示。

图 4.4　在 Wercker 中配置部署对象

来源：wercker.com

图 4.4 显示了一个名为 dockerhub 的部署对象。此对象表示从主分支自动进行部署。这意味着每次提交后，在每个构建结束时都会执行 Wercker 配置文件中定义的 deploy 步骤。

```
deploy:
steps:
  - internal/docker-push:
      username: $USERNAME
      password: $PASSWORD
      cmd: /pipeline/source/gogo-service
      port: "8080"
      tag: latest
      repository: cloudnativego/gogo-service
      registry: https://registry.hub.docker.com
```

`internal/docker-push` 中需要包含的属性在 Wercker 的文档中都有说明。这里需要注意的是，我们为 Docker 镜像提供了一些额外的元数据，例如运行命令、默认端口映射和标签。此外，还指定了部署此镜像的 Docker Hub 仓库（`cloudnativego/gogo-service`）。

另外需要注意的是，获取 Docker Hub 仓库的写入访问权限所需的用户名和密码并没有配置在 `wercker.yml` 文件中，而是替换为在 Wercker 网站中部署目标配置时定义的安全环境变量（见图 4.4）。

除了在成功构建后进行自动部署，我们还可以通过点击网站上的按钮或使用 Wercker CLI 手动将构建工件推送到 Docker Hub 上。

现在，大家应该对 Docker、Wercker 以及如何使用这两种技术实现持续集成有了较深入的理解。

读者练习：创建完整的开发流水线

由于无法访问每个人的 GitHub 账户或 Wercker 账户（大家应该在本章开始时创建了一个 Wercker 账户），所以我们将无法为大家提供完整的开发流水线示例。

然而这对于大家来说是一个理想的契机，可以在进一步深入 Go 编程之前暂停一下，先尝试针对最后几章进行一些实践。作为巩固构建开发流水线知识的练习，希望大家创建自己的端到端的持续集成和交付流水线。

这听起来很复杂，但其实不然。事实上，我们会给各位提供一些思路。

1. 在`$GOPATH/src/github.com/youraccount/pipeline` 中创建一个新的 Go 应用程序（记住，包名为 `main`）。

2. 这个应用程序包含一个 `main.go` 文件，可以输出“hello world”（或任何想

要的内容）到控制台。

3．确保应用程序在本地运行（`go run`）和构建（`go build`）。

4．将代码提交给 GitHub。正常情况下，应该能够在第二台计算机（如果觉得有风险的话，可以在计算机上使用不同工作区）上直接从 GitHub 上 `go get` 刚创建的应用程序。

5．转到 Wercker 并创建一个名为 `mypipeline`（或更具创造性的名称）的新应用程序。选择与 `github` 集成，将其指向公共 GitHub 仓库。

6．完成配置 Wercker 应用程序并跳过 `wercker.yml` 步骤。使用我们之前在讨论 Wercker 时提供的模板。在 `wercker.yml` 中，可以直接使用 Go 命令而不是 `godep`。

7．确保可以使用 Wercker CLI 在本地构建应用程序。

8．为应用程序添加一个推送到 Docker Hub 的部署对象。可以按照前面提到的步骤来执行。

9．对应用程序进行一些小小的改动，并将其提交到 GitHub 中。到 Wercker 中查看构建结果。

10．查看应用程序，然后将它部署到 Docker Hub 上。该程序应该会存在于我们在 `wercker.yml` 文件中选择的位置，例如 `dockerhub.com/(youraccount)/pipeline:latest`。

11．使用之前提供的 `docker run` 命令从 Docker Hub 镜像中执行应用程序。如果一切顺利，则可以看到控制台的输出。

12．提交更改到 GitHub，在几分钟后查看 Docker Hub 中的镜像更改。

以上步骤可能只是为了在控制台中打印一些文本，实际上我们需要关注更高的层面：大家刚刚设置了一个持续交付流水线，它会在每个 GitHub 提交后的几分钟内自动发布新的工件到 Docker Hub 上。

在开始开发项目之前，大家已经拥有了一个稳定、可测试和持续部署的基础环境。从现在开始，如果遵循 TDD 原则和*云之道*，就能通过迭代方式添加任何功能，并确保它可以在签入源代码控制器后部署到生产环境中。

如果想拥有一个为自己所用的基础环境，它将花费远远超过 20 分钟的时间，因此可以先创建一个初始流水线。

高级挑战：集成第三方库

作为一个高级挑战，内容是修改读者练习部分编写的代码，其中要使用一些第三方包（第 3 章中的 table-writer 是一个理想的选择）。

为了完成这个工作，请参阅本章中使用 Glide 工具管理依赖的部分。如果签入了 vendor 目录，并且仿照示例使用了 Glide，那么本地和远程 Wercker 构建应该都能正常工作。

本章小结

本章没有涉及任何 Go 代码。云之道的基础是遵循持续集成和持续交付原则，在当今这个快速变化的世界，我们认为必须采用敏捷开发和云设施，因为它们会在某种程度上帮我们树立对应用程序和服务的信心。

本章讨论了使用 Wercker 作为持续集成首选工具的相关内容，它依赖于 Docker 创建和构建固定的工件。虽然可以自由选择 CI 软件，但建议选择 Wercker，因为它易用、简单且花费很低。换句话说，即使所在的企业选择了其他方式，它也会采用类似于本书中采用的 Wercker 方式建立构建流水线。

我们希望大家已经进行了读者练习，建立构建流水线应该成为一种习惯。在本书的其他部分，每章仍将至少建立一个 Wercker 构建，因此熟悉这个工具是很关键的。

在完成读者练习和后面的高级挑战后，大家应该拥有了核心的 Go 开发技能和基础设施工具，我们需要卷起袖子并深入阅读本书的其他部分，获得使用 Go 完成更复杂更强大的云端应用程序开发的能力。

5

在 Go 中构建微服务

> 黄金法则：你可以在不更改任何其他代码的前提下更改服务并重新部署吗？
>
> ——Sam Newman，*Building Microservices* 作者

我们构建的每个服务都应该是微服务，正如本书前面提到的，我不赞同使用前缀“micro”。本章将要构建一个服务，不只关注结果，也同样关注过程。

我们会采用 **API First** 的方式，在编写代码之前首先设计服务的 RESTful 接口。在开始编写代码时，首先编写测试，通过编写从失败到通过的小测试逐步建立服务。

本章中构建的示例的功能是实现 Go 游戏服务器。此服务允许任何类型的客户参加比赛，不论是通过 iPhone、浏览器还是其他方式。

首先，这个服务需要一个名字。一个使用 Go 实现的服务，又是用于进行 Go 游戏比赛的服务，没有比 **GoGo** 这个名字更适合的了。

本章将介绍以下几个方面。

- API First 开发准则和实践。
- 创建微服务框架。
- 向服务添加测试，并迭代地添加代码，使测试通过。
- 在云端部署和运行微服务。

设计 API First 的服务

在下一节中，我们将设计微服务。软件开发的过程中有一个经典问题：开发结果很少符合最初的设计。文档、需求和实现之间始终存在差距。

幸运的是，下一节大家将看到一些适用于微服务开发的工具，这些工具可以实现设计文档化，并可以最终集成到开发过程中。

设计 match API

为了创建一个管理 match 的服务，要做的第一件事是定义 match 的资源集合。要想使用集合，应该能够创建新的 match 同时列出当前由服务器管理的所有 match，如表 5.1 所示。

表 5.1 match API

资　源	方　法	描　述
/matches	GET	查询所有可用的 match 列表
/matches	POST	创建和开始一个新的 match
/matches/{id}	GET	查询一个 match 的详情

如果开发的是一款需要真实货币购买的 Go 游戏，而不是一个样品，则还需要实现一些方法用来支持在 UI 上查询如 **chains** 和 **liberties** 之类的基本概念，从而确定 Go 中的合法移动。

设计 move API

一旦服务开始管理 match，我们便需要提供一个 API，它可以让玩家进行移动。需要添加以下 HTTP 方法到 moves 子资源中，如表 5.2 所示。

表 5.2 move API

资　源	方　法	描　述
/matches/{id}/moves	GET	返回一个按照时间排序的比赛中所有移动的列表
/matches/{id}/moves	POST	进行移动。移动若没有包含位置信息，则视为略过

创建 API Blueprint

为了简化所做的一切工作，以前我们会尝试抛弃复杂或烦琐的文档。那么我们真的需要分享那些具有苛刻兼容性要求的骇人听闻的文档吗？

对我们来说，Markdown[1]是编写文档和其他文件的首选工具。它是一个简单的纯文本格式，不需要使用 IDE 或其他臃肿的编辑工具，同时可以被转换成 PDF、网页

1 关于 Markdown 的语法参考可以查看以下链接：https://en.wikipedia.org/wiki/Markdown。

等无数的格式。实际上，关于文档格式的争论已经引发了大规模的“血腥”的办公室战争。

我们通常习惯在服务中使用 Markdown 文档。这样能够允许其他开发人员快速获得所有服务的 REST 资源、URI 模式和请求/响应有效荷载的列表。就像 Go 代码一样，我们仍然想要一种简单的方法来记录服务协议，而不是让别人通过路由代码进行查询。

事实证明，有一种 Markdown 的特殊使用方式可以专门用于记录 RESTful API：**API Blueprint**。大家可以在 API Blueprint 网站 https://apiblueprint.org/上阅读关于此格式的资料。

如果查看本章的 GitHub 仓库（https://github.com/cloudnativego/gogo-service），可以看到一个名为 `apiary.apib` 的文件。此文件包含 Markdown 格式的内容，表示 GoGo 服务支持的 RESTful 接口的文档和规范。

代码清单 5.1 为 Markdown 格式的示例，大家可以看到它是如何描述 REST 资源、HTTP 方法和 JSON 有效数据的。

代码清单 5.1 Blueprint Markdown 示例

```
### Start a New Match [POST]

You can create a new match with this action. It takes information about the players
and will set up a new game. The game will start at round 1, and it will be
**black**'s turn to play. Per standard Go rules, **black** plays first.

+ Request (application/json)

        {
            "gridsize" : 19,
            "players" : [
            {
                "color" : "white",
                "name" : "bob"
            },
            {
                "color" : "black",
                "name" : "alfred"
            }
            ]
        }
```

```
+ Response 201 (application/json)

    + Headers

            Location: /matches/5a003b78-409e-4452-b456-a6f0dcee05bd

    + Body

            {
                "id" : "5a003b78-409e-4452-b456-a6f0dcee05bd",
                "started_at": "2015-08-05T08:40:51.620Z",
                "gridsize" : 19,
                "turn" : 0,
                "players" : [
                    {
                        "color" : "white",
                        "name" : "bob",
                        "score" : 10
                    },
                    {
                        "color" : "black",
                        "name" : "alfred",
                        "score" : 22
                    }
                ]

            }
```

通过 Apiary 测试和发布文档

在第 1 章“云之道”中，我们提到过不要过度依赖工具。

工具应该使生活变得更轻松，但它们永远都不应该成为强制性的。包含服务的文档和规范的 API Blueprint Markdown 仅仅是一个简单的文本文件，我们可以使用一个工具使生活变得更加轻松。

Apiary 是一个网站，它能实现以交互方式设计 RESTful API。可以把它想象成支持 API Blueprint Markdown 语法的 WYSIWYG（所见即所得）编辑器，但这只是一个开始。Apiary 可以生成返回 JSON 有效数据的模拟服务器，这样节省了必须自己搭建模拟服务器的时间，并同时允许保持 API First 方式，直到通过制定 API 的草稿阶段。

除提供模拟服务器外，还可以在编写服务器代码之前使用生成的多种编程语言的客户端代码进一步协助团队来验证 API。

GoGo 服务的 API Blueprint 文档可以在 GitHub 仓库以及 Apiary 上进行查看，具体请访问 http://docs.gogame.apiary.io/。本书中不会存储整套文档，我们会把大部分细节保留在 blueprint 文档和 Apiary 上，供各位读者自行阅读。

本章的目的不是教大家如何编写一个游戏服务器，而是让大家学会如何在 Go 语言中构建一个服务。因此，相比测试驱动开发和设置服务框架，一些技术细节（如 Go 语法规则和实际的游戏实现）是相对次要的部分，我们将在后面的章节中介绍。

架设微服务

在理想环境下，可以从一块白板开始直接进入测试。但很少存在理想与完美的世界。在示例中，我们希望能够从 RESTful 接口开始编写测试。

而现实情况是，我们并不能够真正为 RESTful 接口编写测试，除非知道将要为每个接口编写什么样的函数。为了解决这个问题，同时配置服务的基本框架，需要创建两个文件。

第一个文件是 `main.go`（见代码清单 5.2），其中包含主要功能，创建和运行一个新的服务器。这种情况下需要保持主函数尽可能小，因为主函数通常很难单独测试。

代码清单 5.2 main.go

```
package main

import (
  "os"
  service "github.com/cloudnativego/gogo-service/service"
)

func main() {
  port := os.Getenv("PORT")
  if len(port) == 0 {
    port = "3000"
  }

  server := service.NewServer()
  server.Run(":" + port)
}
```

代码清单 5.2 中的代码调用了一个名为 `NewServer` 的函数。此函数返回一个指向 Negroni 结构体的指针。Negroni 是一个第三方库，用于在 Go 内置的 `net/ http`

包上构建路由接口。

注意，粗体的代码行很重要。外部配置对于构建云端应用程序的能力至关重要。允许应用程序从环境变量接受绑定端口，是迈出构建云端运行服务的第一步。一些云提供商也会使用确定的环境变量来自动注入应用程序端口。

代码清单 5.2 为服务器实现程序。这段代码以**经典**模式创建和配置 Negroni，使用 Gorilla Mux 作为路由库。我们通常谨慎地对待任何第三方依赖，必须确保其内部的功能不是 Go 语言的核心库所提供的。

Negroni 和 Mux 可以完美地运行在 Go `net/http` 上，它们是可扩展的中间件，不会干扰将来所做的任何工作。没有事情是强制性的，没有“魔法”，只有一些便利的库使我们不用花费太多时间就可以编写每个服务的模版。

有关 Negroni 的信息，请查看 GitHub repo https://github.com/codegangsta/negroni。有关 Gorilla Mux 的信息，请查看 https://github.com/gorilla/mux 上的仓库。请注意，以上是直接导入代码的地址，非常容易追踪第三方包的文档和源码。

代码清单 5.3 显示了 `main` 函数引用的 `NewServer` 函数和一些实用函数中的内容。请注意，`NewServer` 由于首字母是大写的，所以是可导出的，而 `initRoutes` 和 `testHandler` 则不可以。

代码清单 5.3　server.go

```
package service

import (
        "net/http"

        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
        "github.com/unrolled/render"

)

// NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {

        formatter := render.New(render.Options{
                IndentJSON: true
        })

        n := negroni.Classic()
```

```
        mx := mux.NewRouter()

        initRoutes(mx, formatter)

        n.UseHandler(mx)
        return n
}

func initRoutes(mx *mux.Router, formatter *render.Render) {
        mx.HandleFunc("/test", testHandler(formatter)).Methods("GET")
}

func testHandler(formatter *render.Render) http.HandlerFunc {

        return func(w http.ResponseWriter, req *http.Request) {
                formatter.JSON(w, http.StatusOK,
                 struct{ Test string }{"This  is a test"})
        }
}
```

在这个框架中，最重要的是理解 `testHandler` 函数。与我们一直使用的常规函数不同，此函数返回了一个匿名函数。

此匿名函数反过来返回类型为 `http.HandlerFunc` 的函数，其定义如下。

```
type HandlerFunc func(ResponseWriter, *Request)
```

这种类型定义实际上允许将具有此签名的任何函数都视为 HTTP 处理程序。可以发现此类模式在 Go 的核心包和许多第三方包中被广泛使用。

在简单框架里，我们通过调用 `formatter.JSON` 方法返回一个将匿名结构传递给响应写入器的函数（这就是将 formatter 传递给 wrapper 函数的理由）。

这样做很重要的原因是，所有的 RESTful 接口都是返回 `http.HandlerFunc` 类型函数的包装函数。

在编写测试之前，需要确保框架可以正常工作，以便测试我们实现的资源。可以通过以下命令进行构建（过程可能与 Windows 不同）。

```
$ go build
```

执行以上代码将构建文件夹中的所有 Go 文件。一旦创建了可执行文件，便可以运行 GoGo 服务了。

```
$ ./gogo-service
[negroni] listening on :3000
```

当访问 `http//localhost：3000/test` 时，可以在浏览器中获取 JSON 测试数据。由于使用了 Negroni 的经典配置，所以我们已经实现了一些 HTTP 请求处理功能。

```
[negroni] Started GET /test
[negroni] Completed 200 OK  in 212.121µs
```

框架已经可以正常工作了，Web 服务器至少能够处理简单的请求，现在是时候进行一些真正的测试驱动开发了。

构建 Test First 的服务

只讨论 TDD 是很容易的。尽管有无数的博客和书籍赞颂其优点，却很少有人能规范地使用它，更少有人能完整使用它。使用不完整的 TDD 会导致在 TDD 上花费大量时间和精力，却没有收获代码质量和功能信心的提升。

在本章的这一节中，我们将以 test-first 方式为服务编写一个方法。正常情况下，我们应该会花费 95%的时间编写测试，花费 5%的时间编写代码。测试代码应该明显多于被测试的代码，这是因为我们需要更多的代码来覆盖测试函数所有可能的执行路径，而不只是编写测试函数本身。关于此概念的更多细节，请查阅由 Jez Humble 和 David Farley 所著的 *Continuous Delivery* 一书。

许多组织认为编写测试是在浪费时间，因为它不会增加任何价值，并且实际上会延长发布的时间。这种目光短浅的看法存在很多问题。

TDD 确实会在初期降低开发效率，然而当我们理解一个新的关于团队开发的定义后便不会这样想了。

> Development(n)：*应用程序新增功能特性处于没有上线压力的时期。*
>
> ——Dan Nemeth

考虑以上定义，当审视应用程序的整个生命周期时，会发现应用程序只有很少一部分时期是处在这种“开发”状态下的。

测试的投入将在应用程序的整个生命周期内产生红利，特别是在生产环境中。

- 必须保证正常运行时间。
- 必须及时满足更改/功能需求。

- 调试是奢侈和困难的，有些时候是不可能进行的。

为了开始 TDD 服务的创建之旅，我们首先新建一个名为 handlers_test.go 的文件（如代码清单 5.4 所示）。这个文件将测试 handlers.go 文件中编写的函数。如果文本编辑器支持并行或分屏模式，这将非常有用。

我们将为 POST 请求匹配的 HTTP 处理程序编写一个测试。如果回过头查看 Apiary 文档，会发现此函数在成功时会返回 **201（已创建）**的 HTTP 状态码。

开始编写测试。我们将调用函数 TestCreateMatch，和所有 Go 的单元测试一样，它将作为一个指向 testing.T 结构体的指针。

创建第一个失败测试

为了测试服务器创建 match 的功能，需要调用 HTTP 处理程序。可以通过构建 HTTP 管道的所有组件（包括请求、响应流和头文件等）来手动调用。幸运的是，Go 为我们提供了一个测试 HTTP 服务器，它并不需要开启套接字，但可以让我们调用 HTTP 处理程序。

这里需要完成很多工作，以下是在一次迭代中测试文件的完整代码，与 TDD 思想保持一致，这是一个失败测试。

代码清单 5.4 handlers_test.go

```
package main

import (
        "bytes"
        "fmt"
        "io/ioutil"
        "net/http"
        "net/http/httptest"
        "testing"

        "github.com/unrolled/render"
)

var (
    formatter = render.New(render.Options{
            IndentJSON: true
    })
)

func TestCreateMatch(t *testing.T) {
```

```
    client := &http.Client{}
    server := httptest.NewServer(
        http.HandlerFunc(createMatchHandler(formatter)))
    defer server.Close()

    body := []byte("{\n  \"gridsize\": 19,\n  \"players\": [\n  {\n
    \"color\": \"white\",\n   \"name\": \"bob\"\n   },\n  {\n
    \"color\": \"black\",\n   \"name\": \"alfred\"\n }\n  ]\n}")

    req, err := http.NewRequest("POST",
        server.URL, bytes.NewBuffer(body))
    if err != nil {
        t.Errorf("Error in creating POST request for createMatchHandler: %v", err)
    }
    req.Header.Add("Content-Type", "application/json")

    res, err := client.Do(req)
    if err != nil {
        t.Errorf("Error in POST to createMatchHandler: %v", err)
    }

    defer res.Body.Close()

    payload, err := ioutil.ReadAll(res.Body)
    if err != nil {
        t.Errorf("Error reading response body: %v", err)
    }

    if res.StatusCode != http.StatusCreated {
        t.Errorf("Expected response status 201, received %s", res.Status)
    }

    fmt.Printf("Payload: %s", string(payload))
}
```

使用 Apiary 的另一个原因是，如果查看关于 ***create match*** 功能的文档，点击该方法，它会自动生成 Go 客户端的代码示例。大部分生成的代码都会在代码清单 5.3 前面部分的测试方法中使用到。

第一步是调用 `httptest.NewServer`，它创建了一个 HTTP 服务器监听自定义 URL，并提供指定的方法。之后，使用 Apiary 生成的示例客户端代码来调用此方法。

此处为大家提供两个主要断言。

- 执行请求和读取响应字节时，不会收到任何错误。

- 响应状态码为 ***201（已创建）***。

如果尝试运行以上测试，将看到编译失败。这才是真正的 TDD，因为我们还没有编写测试方法（`createMatchHandler` 还不存在）。为了使编译通过，可以将之前的测试方法添加到 **`handlers.go`** 文件中，如代码清单 5.5 所示。

代码清单 5.5 handlers.go

```
package main

import (
        "net/http"

        "github.com/unrolled/render"
)

func createMatchHandler(formatter *render.Render) http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
                formatter.JSON(w,
                 http.StatusOK,
                 struct{ Test string }{"This is a   test"})
        }
}
```

现在尝试编译。首先，可以输入以下命令进行测试。

```
$ go test -v $(glide novendor)
```

应该能看到以下输出。

```
Expected response status 201, received 200 OK
```

目前，我们写了第一个失败测试！可能仅凭这一点，一些人便会开始怀疑这种方式。请相信我们，本章结束前一定会让大家看到曙光浮现。

为了使测试通过，需要在 HTTP 处理程序处返回 201 状态码。不写完整的实现，也不添加复杂的逻辑，唯一要做的就是让测试通过。这个过程至关重要，我们只写让测试通过的最小代码，如果添加了额外代码，就不再遵循 Test First 模式了。

为了使测试通过，将 formatter 的所在行改为以下形式。

```
formatter.JSON(w, http.StatusCreated, struct{ Test string }{"This is a test"})
```

更改 `http.StatusCreated` 的第二个参数。运行测试，可以看到如下所示的输出。

```
$ go test -v $(glide novendor)
```

```
=== RUN   TestCreateMatch
--- PASS: TestCreateMatch (0.00s)
PASS
ok   github.com/cloudnativego/gogo-service  0.011s
```

测试 Location Header

接下来要编写响应 ***create match*** 的请求（如 Apiary 文档中所述），并在 HTTP 响应中设置 ***Location*** header。按照惯例，当 RESTful 服务新建一些资源时，***Location*** header 应该被设置为新创建资源的 URL。

和往常一样，从一个失败测试开始，然后使测试通过。在测试中添加以下断言。

```
if _, ok := res.Header["Location"]; !ok   {
   t.Error("Location header is not set")
}
```

重新运行测试，结果会失败，同时显示以上错误信息。为了使测试通过，在 `handlers.go` 中修改 `createMatchHandler` 方法，代码如下。

```
func createMatchHandler(formatter *render.Render) http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
                w.Header().Add("Location", "some value")
                formatter.JSON(w, http.StatusCreated,
                        struct{ Test string }{"This  is atest"})
        }
}
```

请注意，我们没有为 Location 添加实际值，相反，只是添加了“some value”。接下来，我们将添加一个会导致失败的测试条件。获取一个包含 `matches` 资源的有效 location header，它有足够的长度，以便知道它还包含新创建 match 的 GUID。修改先前针对 location header 编写的测试，代码如下所示。

```
loc, headerOk := res.Header["Location"]
if !headerOk {
        t.Error("Location header is not set")
} else {
        if !strings.Contains(loc[0], "/matches/") {
                t.Errorf("Location header should contain '/matches/'")
        }

        if len(loc[0]) != len(fakeMatchLocationResult) {
                t.Errorf("Location value does not contain guid of new match")
        }
}
```

我们还在测试中声明了一个名为 fakeMatchLocationResult 的常量，这只是一个字符串，在 Apiary 中也标明了 location header 的测试值。下面将使用它来测试断言和模拟值。定义如下。

```
const(
   fakeMatchLocationResult ="/matches/5a003b78-409e-4452-b456-a6f0dcee05bd"
)
```

壮丽的蒙太奇：迭代测试

由于本书篇幅有限，因此不会提供在测试迭代期间从红灯（失败）到绿灯（通过）更改的所有代码。

我们只会描述为了使 TDD 通过而进行的工作。

- 编写一个失败测试。
- 使失败测试用例通过。
- 检查结果。

如果想查看操作历史，可以逐行翻阅我们在 GitHub 中的提交历史，以获取每一行的代码更改。搜索所有被标记为“TDD GoGo service Pass *n*”的提交，其中 *n* 是测试迭代次数。

总结针对每个失败测试所采用的方法，建议大家伴随着好莱坞黑客电影的蒙太奇背景音乐，阅读以下内容。

1. **TDD Pass 1**。我们创建了测试 HTTP 服务器所需的初始配置，该服务器调用 HTTP 处理程序方法（被测方法）。由于被测方法尚不存在，测试开始时会编译失败。将测试资源代码添加到 createMatchHandler 方法中可以使测试通过。

2. **TDD Pass 2**。添加断言，判断 HTTP 返回值中是否包含 ***Location*** header。最初测试失败，因此在 location header 中添加了一个占位符。

3. **TDD Pass 3**。添加断言，***Location*** header 是一个正确格式的 URL，指向由 GUID 标识的 match。最初测试失败，随后生成一个新的 GUID 和设置正确的 location header。

4. **TDD Pass 4**。添加断言，判断 HTTP 返回值中 match 的 ***ID*** 和 location header 中的 GUID 是否相等。最初测试失败，为了通过测试，需要在测试端添加能解析返回数据的代码。这意味着必须创建一个可以在服务器端返回有效数据的结构体。因此不在处理程序中返回“this is a test”，而是返回一个真正的响应对象。

5. **TDD Pass 5**。添加断言，判断被处理函数使用的存储库是否已经包括了新创建的 match。为此，必须创建一个存储库接口并实现一个内存存储库。

6. **TDD Pass 6**。添加断言，判断服务返回的 match 大小和存储库中的是否相同。这使得我们必须为响应创建一个新的结构体，并进行更新。除此之外，我们还更新了另一个 gogo-engine 库，它实现了 Go 游戏需要的最小分辨率的业务逻辑，应该在最大程度上与 GoGo 服务解耦。

7. **TDD Pass 7**。添加断言，判断创建 match 请求时包含的游戏玩家和服务端的 JSON 返回中的值是否相同，并且是否也相应地保存在存储库中。

8. **TDD Pass 8**。添加断言，测试如果发送除 JSON 格式以外的数据，或者没有为创建 match 请求发送合理的值，服务器是否会返回 ***"Bad Request"***。这些断言会失败，因此需要为处理程序添加检验 JSON 格式和无效请求对象的代码。Go 可以很好地支持 JSON 反序列化，所以通过检查反序列结构体中缺少的变量或默认值可以捕获大多数“bad request”输入。

让我们拭目以待，看看在这组迭代之后会发生什么变化。代码清单 5.6 显示了使用 TDD 开发的一个处理程序，迭代测试失败，然后编写代码通过测试。值得一提的是，我们不会编写任何代码，除非它可以使测试通过。这在最大程度上保证了测试和可信度的覆盖。

对于许多开发人员和组织来说，这是艰难的一步，但这是值得的。这种开发方式给很多部署在云端的真实应用程序带来了非常多的帮助。

代码清单 5.6 handlers.go （8 轮 TDD 迭代后）

```
package service

import (
        "encoding/json"
        "io/ioutil"
        "net/http"

        "github.com/cloudnativego/gogo-engine"
        "github.com/unrolled/render"
)

func createMatchHandler(formatter *render.Render, repo matchRepository)
     http.HandlerFunc {
        return func(w http.ResponseWriter, req *http.Request) {
        payload, _ := ioutil.ReadAll(req.Body)
```

```
        var newMatchRequest newMatchRequest
        err := json.Unmarshal(payload, &newMatchRequest)
        if err != nil {
          formatter.Text(w, http.StatusBadRequest,
            "Failed to parse create match  request")
          return
        }
        if !newMatchRequest.isValid() {
          formatter.Text(w, http.StatusBadRequest,
           "Invalid new match request")
          return
        }

        newMatch := gogo.NewMatch(newMatchRequest.GridSize,
         newMatchRequest.PlayerBlack, newMatchRequest.PlayerWhite)
        repo.addMatch(newMatch)
        w.Header().Add("Location", "/matches/"+newMatch.ID)
        formatter.JSON(w, http.StatusCreated,
          &newMatchResponse{ID: newMatch.ID,
                    GridSize: newMatch.GridSize,
                     playerBlack: newMatchRequest.PlayerBlack,
                    PlayerWhite: newMatchRequest.PlayerWhite})
        }
}
```

虽然 Go 编码指南建议采用包含 8 个字符的 tab，但实际上进行压缩后可以使内容更加可读。

单个函数中包含了大约 20 行代码，该函数的两个测试函数中大约有 120 行代码。这正是我们希望达到的比率。甚至在使用 HTTP 测试工具测试服务之前，我们就希望拥有 100%的信心，并清楚地知道服务应该如何运行。

基于以上编写的测试和代码清单 5.6 中的代码，大家能发现任何测试缺陷吗？能否找到任何可能绕过代码的场景或边界情况是我们还未在测试里考虑到的？

对此，我们发现了以下两点。

1．此服务不是无状态的。如果服务宕机，所有进行中的游戏将丢失。这是一个已知的问题，只能顺其自然。在第 7 章中，我们将解决数据持久性问题。

2．存在无法监测的滥用情况。值得注意的是，我们无法阻止玩家一个接一个地不断快速创建游戏，直到超出内存容量，最后使服务崩溃。发生这种特殊滥用情况的原因在于使用内存存储游戏，这违背了云的最重要的原则：无状态。针对这一问题，无须编写测试，因为正如第 1 章中提到的，这些代码是临时的，我们要避免编

写 DDoS 代码。

我们会在本书中不断更正一些内容，但是类似防范边界攻击的这类情况，应该是大家在建立企业级服务时考虑的。

在云端部署和运行

现在我们已经使用 Go 构建了一个微服务，同时遵循云之道，我们可以在云端很好地使用和部署服务。首先要做的是寻找一个云环境。虽然有许多选择，但在本书中，我们选择 Cloud Foundry 的 PCF Dev 和 Pivotal Web Services（PWS）作为部署对象，因为它们都非常容易上手，并且 PWS 有一个试用版本，无须绑定信用卡即可使用。

创建 PWS 账户

访问 http://run.pivotal.io/，使用 Pivotal Web Service 创建账户。Pivotal Web 服务由 Cloud Foundry 提供平台支持，可以实现在云端部署应用程序，并使用其市场中的一些免费和付费服务。

创建账户并登录后，可以看到所在组织的信息中心。组织是安全和部署的逻辑单元，可以邀请其他人加入自己的组织，以便在云项目中进行协作，或完成其他工作。

在组织的首页或资讯主页上，可以看到一个包含有用信息的方块，其中有指向 *Cloud Foundry CLI* 的链接。这是一个命令行接口，可以使用它在任何 Cloud Foundry（而不只是 PWS）中推送和配置应用程序。

下载并安装 CF CLI，运行一些测试命令（如 `cf apps` 或 `cf spaces`）以验证其是否已连接并正常工作。请记住，PWS 有 60 天的试用期，在此期间不必绑定信用卡，因此请合理使用。

有关使用 CF CLI 的详细信息，请参阅 http://docs.run.pivotal.io/devguide/cf-cli/中的文档。

配置 PCF 开发环境

如果可以承担风险，或者只是想简单地做一些小改动，那么可以使用 **PCF Dev**。

本质上讲，**PCF Dev** 是 Cloud Foundry 的一个简化版本，为应用程序开发人员提供了将应用程序部署到 CF 上所需的全部基础功能，但不包含所有生产级别的功能。PCF Dev 可以实现将云运行在自己的笔记本中。

PCF Dev 利用虚拟化（可以在 VMware 或 VirtualBox 之间进行选择）和一个名为 ***vagrant*** 的工具来启动一台独立的虚拟机，该虚拟机将作为 PCF Dev 和应用程序的托管主机。

可以在本地使用 PCF Dev 测试应用程序在云端的运行情况，而无须推送到 PWS 上。我们发现，它对于如服务绑定、自动化集成测试和完全验收测试是非常有帮助的。

在编写本书的时候，PCF Dev 仍处于早期阶段，因此现在的安装和配置说明可能会有所改动。

想了解更多关于 PCF Dev 的信息，请访问 https://docs.pivotal.io/pcf-dev/。

PCF Dev 的优点是，一旦完成安装，就可以简单地发出启动指令，它会在本地虚拟化环境中提供我们所需的一切。例如在 OS X 上，可以使用 `./start-osx` 脚本开始创建基础环境。

使用相同的 Cloud Foundry CLI，可以将它指向 MicroPCF 环境。

```
$ cf api api.local.pcfdev.io --skip-ssl-validation
Setting api endpoint to api.local.pcfdev.io...
OK

API endpoint: https://api.local.pcfdev.io (API version: 2.44.0)
Not logged in. Use 'cf login' to log in.
```

确保按照提示登录（默认用户名和密码为 ***admin*** 和 ***admin***），可以使用 Cloud Foundry CLI 标准命令与本地新创建的私有 CF 通信。

```
$ cf apps
Getting apps in org local.pcfdev.io-org / space kev as admin...
OK
```

提交到 Cloud Foundry

现在已经安装了 CF CLI，并且可以选择让 CLI 以 PWS 云或本地 PCF Dev 环境为目标，下面便可以将应用程序推送到云端运行了。

虽然可以通过手动方式配置所有选项将应用程序推送到云端，但是采用创建

manifest 的方式会更简单一些（随后会进行更多配合 CD 管道的工作），如代码清单 5.7 所示。

代码清单 5.7 manifest.yml

```
applications:
- path: .
  memory: 512MB
  instances: 1
  name: your-app-name
  disk_quota: 1024M
  command: your-app-binary-name
  buildpack: https://github.com/cloudfoundry/go-buildpack.git
```

将此 manifest 文件放在应用程序的主目录中，只需键入以下命令，应用程序便会部署在云端。

```
$ cf push
```

正如我们将在本书后面提到的，甚至可以配置 Wercker 管道，以便在持续交付的构建成功时自动将应用程序部署到我们所选择的 Cloud Foundry 中。

> **关于 Go buildpack**
>
> buildpack 旨在将应用程序代码与运行应用程序所需的基础环境相整合。Java buildpack 包含 JDK 和 JRE，Node buildpack 包含 node 等。但是 Go buildpack 太容易违背“单一不可变部件”原则，可能会有人向 buildpack 提交一个破坏代码或管道的更改。正如本书后面将讨论的，当我们部署真正的应用程序时，更倾向于将 Docker 镜像直接从 Docker Hub 部署到云端。至于选择 buildback 还是 Docker 完全取决于个人和公司，这通常是一种个人喜好。

本章小结

在本章中，我们向大家介绍了关于在 Go 中构建微服务的基础知识。讨论了一些建立基础路由和处理函数的代码，更重要的是，介绍了如何构建代码的测试。

此外，我们还将代码部署到了云端。本书的其余部分将涉及更多的技术细节，探索更深入的主题，所以希望大家在继续后面的学习之前，花一点时间来回顾本章中难以理解的内容。

现在是进行一些小的调整并创建自己的 ***hello world*** 服务的好时机，可以将它们部署到 PWS 上，启动、停止扩展应用程序。还可以浏览 PWS 中的市场，了解部署在其中的一些强大的应用程序，包括数据库、消息队列和监控等。

6 运用后端服务

将业务推到云上，再把注意力放在自己的技术核心上。

——Tom Cochran，美国国务院平台副协调员

到目前为止，我们已经成功构建了一个服务，但是在真实场景中往往不仅仅是要构建一个只能发送 JSON 字符串“hello，world”的服务而已。真实场景中会有一些彼此相互依赖的服务，以及一些完整的相互依赖且相互关联的服务系统。

本章将探讨那些与设计服务系统相关的技术。研究如何以 test-first 的方式构建服务系统，测试单个服务，展示如何在互相交互的服务上使用 TDD（测试驱动开发），同时也将介绍一些用于处理跨多个服务节点共享数据的设计模式。

最后，本章将会在云端部署多个服务，并讨论各种关于外部配置和用于彼此交互的服务发现技术。

本章将会涵盖以下内容。

- 设计服务系统。
- 怎样让 TDD 适用于互相依赖的后端服务。
- 用于在服务之间共享数据的设计模式。
- 服务绑定与外部配置。
- 运行时动态服务发现。

设计服务系统

虽然听起来很简单，但实际上服务很少存在于独立隔绝的环境中。服务需要具备授权、认证、数据持久化存储、短期缓存、文档与组件生成器、hoozit 工厂等因素。

服务需要很多组件，一定要保持这样的思路去开发和构建服务，因为如果在一个独立隔绝的环境下开发服务，势必会浪费很多时间在生产环境中来回切换。

不幸的是，图 6.1 展示了许多人心中服务系统的样子。

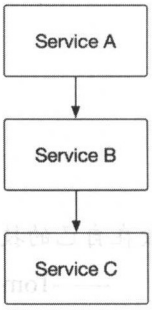

图 6.1　一个不合理的服务依赖关系

在这种不可扩展的方式中，每个服务并不依赖于其他服务，并以特有的方式进行扩展。它有一个明确的头部、底部和一个明显的依赖关系链。

这种方式误导了我们。假设服务 C、B、A 可以依次正常工作，每个服务只与另一个服务进行交互。这样想未免也太简单了。

图 6.1 的层次结构存在着另一个问题，因为很少有来自下游的错误，因此假设这个下游的服务是可以正常工作的，这样的假设很可能会存在漏洞。如果严格遵守云服务的方式进行测试，那么就要测试所有的服务，这样做是完全不会存在漏洞的。

图 6.2 展示了一个更合理的服务架构。虽然看起来有些复杂，但已经比许多企业级的服务系统简单得多。

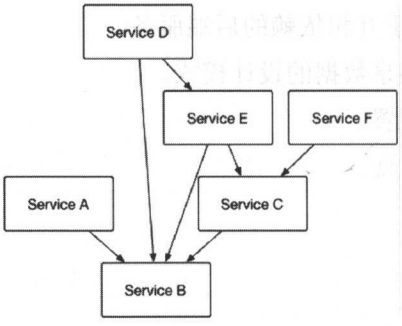

图 6.2　一个更合理的服务依赖关系

怎么去建立这样的服务呢？从方法论的角度来看，要遵循 *API First* 原则。防止图 6.2 中的系统变成难以管理且混乱的唯一方法就是严格遵守 API 协议，并且明确相应的发布周期和版本控制。

在本章的其他部分中，大家将会看到一些具体的构建服务的代码技巧，这些技巧可以在不牺牲测试质量的前提下提前构建测试，也将解答以下问题：服务如何彼此发现？相关的凭证以及其他的配置存在哪里？

以 test-first 方式构建依赖服务

上一章中介绍了以测试驱动开发的方式构建服务所需的所有步骤和要求。实际上就是提前写出运行失败的测试用例，然后编写足够多的代码将它们转变成可以通过的测试，一旦习惯养成后，慢慢就会形成自然。

以这种方式编写了多年的代码之后，再用其他的方式编写代码就会感觉很不自然，最主要的是不可靠。如果没有事先编写测试用例，谁也不会对自己编写的代码保有信心，不管自己有多厉害或写了多少本书。

这个习惯不仅可以提高测试质量，还会带给我们自信。任何人都可以编写高覆盖率的测试，但需要多一点的思考和良好的习惯，写测试会带来信心并可以让应用程序正确地执行。

本章将构建一个服务，然后再构建另一个依赖于它的服务。一个服务依赖于另一个服务听起来并没有什么稀奇，但是比较特殊的是要在服务的请求处理器中去调用另一个服务。

我们要构建一些 test-first 且松耦合的示例服务：*catalog* 服务和 *fulfillment* 服务。catalog 服务是一个运行着网站 Web UI 的服务，而 fulfillment 服务是一个仓库服务，显示当前仓库的商品剩余量以及分配商品的时间。

在本示例中，catalog 服务返回商品的详细信息，也能够通过查询 fulfillment 服务获得商品信息来扩充数据集并返回给调用者。这个例子过于简单，有点类似图 6.1 所示的结构。以正确的方式去构建这个示例可以帮助我们在今后正确地构建真实且复杂的微服务系统。

构建 fulfillment 服务

接下来的第一步就是构建 fulfillment 服务。之所以从这个服务开始是因为它并不依赖其他服务。若是对于真实的应用程序，就不会这样轻松了。如果有依赖服务，可能就需要凭借 API 协议让多个服务去严格遵守这个协议进行并行构建。

示例中所有的服务均采用相同的项目结构，包括一个 `main.go` 文件和一个 service 子包。这个 ***service*** 包中含有一个 `server.go` 文件，一个 `handlers.go` 文件，和一个里面写有 HTTP 请求处理器测试用例的 `handlers_test.go` 文件。具体可以参考 fufillment 服务的 GitHub 仓库：https://github.com/cloudnativego/backing-fulfillment。

由于采用了 TDD，所以第一件事就是编写测试用例，代码清单 6.1 完整地展示了这些测试用例（源码中的制表符已经被更改，以便读者更好地阅读）。

代码清单 6.1 handlers_test.go

```
package service

import (
        "encoding/json"
        "io/ioutil"
        "net/http"
        "net/http/httptest"
        "testing"

        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
        "github.com/unrolled/render"
)

var (
        formatter = render.New(render.Options{
                IndentJSON: true,
        })
)

func TestGetFullfillmentStatusReturns200ForExistingSKU(t *testing.T) {
        var (
                request  *http.Request
                recorder *httptest.ResponseRecorder
        )

        server := NewServer()
```

```
    targetSKU := "THINGAMAJIG12"

    recorder = httptest.NewRecorder()
    request, _ = http.NewRequest("GET", "/skus/"+targetSKU, nil)
    server.ServeHTTP(recorder, request)

    var detail fulfillmentStatus

    if recorder.Code != http.StatusOK {
      t.Errorf("Expected %v; received %v", http.StatusOK, recorder.Code)
    }
    payload, err := ioutil.ReadAll(recorder.Body)
    if err != nil {
      t.Errorf("Error parsing response body: %v", err)
    }
    err = json.Unmarshal(payload, &detail)
    if err != nil {
      t.Errorf("Error unmarshaling response to fullfillment status: %v", err)
    }

    if detail.QuantityInStock != 100 {
      t.Errorf("Expected 100 qty in stock, got %d", detail.QuantityInStock)
    }
    if detail.ShipsWithin != 14 {
      t.Errorf("Expected shipswithin 14 days, got %d", detail.ShipsWithin)
    }
    if detail.SKU != "THINGAMAJIG12" {
      t.Errorf("Expected SKU THINGAMAJIG12, got %s", detail.SKU)
    }
}
```

本例中有以下几个测试断言。

- 从资源`/skus/{sku}`中接收一个 ***200*** 状态码。
- 可以解析这个响应体。
- 响应体可以被解析成 `fulfillmentStatus` 结构体。
- 构造一个假的响应体，因为还没有构建出一个功能完善的服务。

回顾上一章，像这样的测试并不是一气呵成的。而是要编写一个运行失败的测试断言后，通过代码实现让它运行通过，然后再编写另一个测试断言，再通过代码实现让它运行通过，这样不断重复才构建完成的。

依照这种方式构建测试，可以让我们从繁重的代码迭代检查中解脱出来。

fulfillment 服务只包含一个请求处理器，它的代码如代码清单 6.2 所示。不要忘记我们现在测试的是故意用伪造的虚假数据作为返回值的服务。如果想将其变成更符合生产的功能性服务，首先要做的就是编写测试断言，证明此服务没有使用虚假数据作为返回值。如果是虚假的数据，那么这些测试将会运行失败。所以，如果想让测试运行通过，就不得不编写功能代码让服务显得更“真实”。

代码清单 6.2 handlers.go

```
package service

import (
  "net/http"

  "github.com/gorilla/mux"
  "github.com/unrolled/render"
)

// getFullfillmentStatusHandler simulates actual fulfillment by supplying
// bogus values for QuantityInStock
// and ShipsWithin for any given SKU. Used to demonstrate a backing service
// supporting a primary service.
func getFullfillmentStatusHandler(formatter *render.Render) http.HandlerFunc {
 return func(w http.ResponseWriter, req *http.Request) {
        vars := mux.Vars(req)
        sku := vars["sku"]
        formatter.JSON(w, http.StatusOK, fulfillmentStatus{})
                SKU:             sku,
                ShipsWithin:     14,
                QuantityInStock: 100,
        })
 }
}
func rootHandler(formatter *render.Render) http.HandlerFunc {
         return func(w http.ResponseWriter, req *http.Request) {
                 formatter.Text(w,
                  http.StatusOK,
                  "Fulfillment Service, see (url) for API.")
         }
}
```

这个函数通过返回仅包含一个变量值 SKU 的虚假值来使测试通过。这个返回值信息包含两周内的交易商品和 100 个商品库存。

最后，需要设置路由才能让浏览器或者 RESTful 客户端进行测试，请看如下的

initRoutes 方法。

```
func initRoutes(mx *mux.Router, formatter *render.Render) {
        mx.HandleFunc("/skus/{sku}",
        getFullfillmentStatusHandler(formatter)).Methods("GET")
}
```

要相信 fulfillment 模拟器可以按照预期正常工作，因为我们已经运行过测试用例了。如果想要运行出如下结果（发送一个 SKU 参数值为 WIDGET42 的 GET 请求），就要采用如下命令（假设目前已经从 Git 上获取了最新的代码并用 glide install 命令安装了相应的第三方包）。

```
$ go build
$ ./fulfillment-service
[negroni] listening on :3001
[negroni] Started GET /skus/WIDGET42
[negroni] Completed 200 OK in 163.614µs
```

构建 catalog 服务

现在已经有了一个 fulfillment 模拟服务，这个服务已经通过了手动单元测试并有很好的代码覆盖率，接下来就可以继续构建 catalog 服务了。

catalog 服务也是一个返回伪造虚假值的模拟器，但是有一个例外：一些数据从 catalog 服务返回后需要再去请求 fulfillment 服务。

发送一个 GET 请求到 catalog 服务并获得 SKU 的值，然后 catalog 服务还会再去请求 fulfillment 服务进而获得这个 SKU 的值并返回相应的伪造数据。

代码清单 6.3 显示了 catalog 服务的测试用例，它和 fulfillment 的测试用例相似，这些测试都是基于期望的伪造值编写的。

代码清单 6.3 handlers_test.go

```
package service
import (
  "encoding/json"
  "io/ioutil"
  "net/http"
  "net/http/httptest"
  "testing"
  "github.com/codegangsta/negroni"
  "github.com/unrolled/render"
)
```

```
var (
  formatter = render.New(render.Options{
        IndentJSON: true
  })
)

func TestGetDetailsForCatalogItemReturnsProperData(t *testing.T) {
  var (
        request  *http.Request
        recorder *httptest.ResponseRecorder
  )

  server := MakeTestServer()

  targetSKU := "THINGAMAJIG12"

  recorder = httptest.NewRecorder()
  request, _ = http.NewRequest("GET", "/catalog/"+targetSKU, nil)
  server.ServeHTTP(recorder, request)

  var detail catalogItem

  if recorder.Code != http.StatusOK {
        t.Errorf("Expected %v; received %v", http.StatusOK, recorder.Code)
  }
  payload, err := ioutil.ReadAll(recorder.Body)
  if err != nil {
        t.Errorf("Error parsing response body: %v", err)
  }
  err = json.Unarshal(payload, &detail)
  if err != nil {
        t.Errorf("Error unmarshaling response to catalog item: %v", err)
  }
  if detail.QuantityInStock != 1000 {
        t.Errorf("Expected 100 qty in stock, got %d", detail.QuantityInStock)
  }
  if detail.ShipsWithin != 99 {
        t.Errorf("Expected shipswithin 14 days, got %d", detail.ShipsWithin)
  }
  if detail.SKU != "THINGAMAJIG12" {
        t.Errorf("Expected SKU THINGAMAJIG12, got %s", detail.SKU)
  }
  if detail.ProductID != 1 {
        t.Errorf("Expected product ID of 1, got %d", detail.ProductID)
  }
}
```

```
func MakeTestServer() *negroni.Negroni {
  fakeClient := fakeWebClient{}
  return NewServerFromClient(fakeClient)
}

type fakeWebClient struct{}

func (client fakeWebClient) getFulfillmentStatus(sku string)
  (status fulfillmentStatus, err error) {
  status = fulfillmentStatus{
       SKU:             sku,
       ShipsWithin:     99,
       QuantityInStock: 1000,
  }
  return status, err
}
```

若要让上面的测试通过，需要创建请求处理器，该处理器使用一个真实的客户端去消费 fulfillment 服务，或者使用一个假的客户端仅仅返回伪造的数据。

代码清单 6.4 为具体的实现代码。

代码清单 6.4 handlers.go

```
package service
import (
        "fmt"
        "net/http"

        "github.com/gorilla/mux"
        "github.com/unrolled/render"

)

// getAllCatalogItemsHandler returns a fake list of catalog items
func getAllCatalogItemsHandler(formatter *render.Render) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
        catalog := make([]catalogItem, 2)
        catalog[0] = fakeItem("ABC1234")
        catalog[1] = fakeItem("STAPLER99")
        formatter.JSON(w, http.StatusOK, catalog)
  }
}

// getCatalogItemDetailsHandler returns a fake catalog item. The key takeaway here
// is that we're using a backing service to get fulfillment status for the individual
// item.
func getCatalogItemDetailsHandler(formatter *render.Render,
```

```
 serviceClient fulfillmentClient) http.HandlerFunc {
 return func(w http.ResponseWriter, req *http.Request) {
        vars := mux.Vars(req)
        sku := vars["sku"]
        status, err := serviceClient.getFulfillmentStatus(sku)
        if err == nil {
                formatter.JSON(w, http.StatusOK, catalogItem{
                        ProductID:      1,
                        SKU:            sku,
                        Description:    "This is a fake product",
                        Price:          1599, //$15.99
                        ShipsWithin:    status.ShipsWithin,
                        QuantityInStock:  status.QuantityInStock,
                })
                } else {
                        formatter.JSON(w, http.StatusInternalServerError,
                      fmt.Sprintf("Fulfillment Client error: %s", err.Error()))
                }
        }
}

func rootHandler(formatter *render.Render) http.HandlerFuxnc {
       return func(w http.ResponseWriter, req *http.Request) {
          formatter.Text(w, http.StatusOK,
  "Catalog Service, see http://github.com/cloudnativego/backing-catalog for API.")
        }
}

func fakeItem(sku string) (item catalogItem) {
        item.SKU = sku
        item.Description = "This is a fake product"
        item.Price = 1599
        item.QuantityInStock = 75
        item.ShipsWithin = 14
        return
}
```

如上所示，数据封装中的两个值 `ShipsWithin` 和 `QuantityInStock` 明确地被后端服务赋值。这个测试通过后，就可以继续下一步测试了。

提到测试，可以采用的测试模式有许多。比如使用注入库和魔法库去创建虚假对象，然后虚假对象记录被调用和没被调用的内容，并且可以在给定的环境下指定虚假对象返回什么值。

上面所说的方法没有错误，但是实现这样的方法可以有很多种方式，如果利用 Go 语言接口的天生优势来做这件事，那么只需要一个接口就够了，这个称之为

will-it-blend typing[1]

接下来要做的是创建 fulfillment 客户端，这个客户端将会发送一个 HTTP 请求到后端服务。如果是创建方便测试的假客户端，那么只需要返回合适的虚假值即可。

代码清单 6.6 展示了一个已被实现的 HTTP 客户端。这里定义了一个 `fulfillmentClient` 接口，该接口只有一个方法供真客户端或假客户端来实现。

Apiary 和客户端代码生成器

调用 HTTP 方法并保存返回的结果是需要经常进行的繁重工作。使用 Apiary 可以使编写 API 代码变得更简单，可以预览并执行示例代码来访问特定的 API。这样的话用任何语言（包括 Go）来生成客户端包装器都是非常方便的。

代码清单 6.6 展示了创建一个 HTTP 请求并将这个请求的返回值反序列化成 `fulfillmentStatus` 结构体的过程。

代码清单 6.6 fulfillment-client.go

```
package service

import (
        "encoding/json"
        "fmt"
        "io/ioutil"
        "net/http"
)

type fulfillmentClient interface {
        getFulfillmentStatus(sku string) (status fulfillmentStatus, err error)
}
type fulfillmentWebClient struct {
        rootURL string
}
func (client fulfillmentWebClient) getFulfillmentStatus(sku string)
 (status fulfillmentStatus, err error) {
        httpclient := &http.Client{}

        skuURL := fmt.Sprintf("%s/%s", client.rootURL, sku)
        fmt.Printf("About to request SKU details from backing service: %s\n", skuURL)
        req, _ := http.NewRequest("GET", skuURL, nil)
```

1 正如前面所讨论的，will-it-blend typing 是一个比“鸭子类型”更准确的关键字，鸭子类型来自以下这段话：“如果一个东西像鸭子一样走路、嘎嘎叫，那它就是一只鸭子。”

```
        resp, err := httpclient.Do(req)
        if err != nil {
                fmt.Printf("Errored when sending request to the server: %s\n",
                 err.Error())
                return
        }

        defer resp.Body.Close()
        payload, _ := ioutil.ReadAll(resp.Body)

        err = json.Unmarshal(payload, &status)
        if err != nil {
                fmt.Println("Failed to unmarshal server response.")
                return
        }
        return status, err
}
```

如下面这段代码所示，后端服务的 URL 都是硬编码的。

```
func NewServer() *negroni.Negroni {

        formatter := render.New(render.Options{
                IndentJSON: true
        })
        n := negroni.Classic()
        mx := mux.NewRouter()
        webClient := fulfillmentWebClient{
                rootURL: "http://localhost:3001/skus",
        }

        initRoutes(mx, formatter, webClient)

        n.UseHandler(mx)
        return n
}

func initRoutes(mx *mux.Router, formatter *render.Render, webClient fulfillmentClient) {
        mx.HandleFunc("/catalog",
          getAllCatalogItemsHandler(formatter)).Methods("GET")
        mx.HandleFunc("/catalog/{sku}",
          getCatalogItemDetailsHandler(formatter, webClient)).Methods("GET")
}
```

若要修复这个硬编码，可以通过读取环境变量的方式在端口 3001 上启动

fulfillment 服务，然后再启动 catalog 服务。在命令终端访问 catalog 服务将会返回如下输出结果。

```
[negroni] Started GET /catalog/THINGY
[negroni] Completed 200 OK in 2.122473ms
```

如果在终端上同样可以看到 fulfillment 服务的输出，那么就证明 catalog 服务的确调用了 fulfillment 服务。

```
[negroni] Started GET /skus/THINGY
[negroni] Completed 200 OK in 79.226µs
```

仔细思考一下，到目前为止我们已经完成了以下步骤。

- 运用 TDD 方式构建了一个依赖于其他服务的服务。
- 构建了另一个 test-first 的服务，此服务返回虚假值，并模仿后端服务的功能。
- 测试通过后就可以创建抽象服务客户端。在代码中采用真实的 HTTP 客户端，在测试代码中使用假数据，以上操作若没有代码生成器、模拟器等工具都是无法完成的。
- 启动所有的服务，每个服务绑定不同端口，确保在调用 catalog 服务时会再访问调用 fullfillment 服务。

在服务之间共享结构化数据

本章已经介绍了一些服务，这些服务访问其他服务暴露出来的数据。在示例中，fulfillment 服务对外暴露一个很小的结构体，这个结构体通过变量 SKU 来表示 fulfillment 服务的库存状态。库存状态信息包括商品的具体交易时间和库存余量。

仔细观察会注意到一些细节，如果不能很好地处理这些细节将直接导致优雅的代码变成肮脏的代码，甚至可能会变成可怕的代码。

fulfillment 服务需要维护一个结构体，这个结构体代表了服务的状态，可以被服务操作并序列化成 JSON 数据。catalog 服务也要维护一个结构体，这个结构体代表了同样的服务状态，所以需要读取 fulfillment 服务暴露出来的 JSON 数据，并将其填充到 catalog 商品的详细结果中。

目前存在一个核心问题，那就是数据模型的共享问题。通常可以用三种方法来解决这个问题，下面将会逐个探讨每一种方法。

客户端引用服务端包

这个解决方案将结构体 `fulfillmentStatus` 变成了 `FulfillmentStatus`（由此变成可导出类型），这样就可以在 catalog 服务中直接读取 fulfillment 服务的状态数据了。

表面上来看这是一个好主意，可以最大化复用代码，没有一行重复的代码，这样就不会因为在 fulfillment 与 catalog 中定义了相同的示例而产生偏差。

这看起来像是一个不错的解决方案，但实际上却存在着危险的副作用。最大的副作用产生在读取 fulfillment 服务暴露出来的所有数据这一环节。现在 catalog 服务与 fulfillment 服务紧密耦合在一起，如果 fulfillment 服务发生某些更改，虽然没有更改公共约定，但也很可能会导致 catalog 服务的编译或运行失败。

所以要尽可能避免使用这种方法。作为作者与开发者，我们一起经历了整个服务系统的开发，让客户端直接引用服务端的代码不会带来任何益处，有时候会导致严重后果。

客户端复制服务端结构

本章代码示例中已经采用了这个方法。在这种情况下，fulfillment 与 catalog 服务都会自定义 `fulfillmentStatus` 结构体并附带 JSON 序列化标签。

许多的开发者和架构师看到这种做法后，估计要产生杀人的冲动了。怎么敢复制粘贴同样的代码两次呢？超过 10 行的重复代码是不可能被允许的，并且会被其他人厌恶。

> **重复代码**
>
> 信不信由你，我们曾经讨论过很多次关于重复代码行数的问题，到底是 10 行还是 3 行。很多人通常会在忘记某种特定模式真正意图的情况下而严格遵守这个行数要求。

若不想改变 fulfillment 服务的内部代码，那就必须改变 catalog 服务的内部代码。即使 catalog 服务采用最新版本的 fulfillment 服务中的结构体（取决于是否关注相关依赖的更新），但是只要 fulfillment 服务改变了自己的内部代码，那么 catalog 服务也只能跟着改变。

概括一下，服务端与客户端可以有完全不一样的内部结构定义，它们都是 API

返回数据的组成部分。这样服务端与客户端可以随意改变自己的内部代码并维持自己的版本发布节奏。举个例子，可以用不同的结构体来组成相同的 JSON 结构。想要获得什么样子的 JSON 数据结构取决于这个数据结构被如何处理，以及这个数据结构是否会被其他服务影响。

为了加强理解，这里引用 Sam Newman 所著的 *Building Microservices* 中的一句话：

> 各服务之间的重复代码远远比服务内部的重复代码糟糕得多。
>
> —— Sam Newman

客户端与服务端引用共享包

现在来看第三种方法。客户端直接引用服务端代码包的这种方式是不理想的，这将会导致丑陋的代码片段产生。通过下面这种方式可以解决这个问题：从客户端和服务端中提取共享数据结构，将它们移动到共享和中立的地方供所有的服务引用。这是一个不错的解决办法，对吗？

不，这种方法具有第一种方法的所有缺点，并成功地伪装成了一种优雅的解决方案。这种方法只是玩了一个 shell 游戏，聪明地结合公有结构体与私有结构体，将坏代码的气味隐藏起来，让多个服务间共享依赖。经验告诉我们，这种方法是不可取的。

应该避免使用这种方法。创建一个真正的敏捷和协作的服务生态系统，唯一安全的方法就是保证它们尽可能地松耦合，这意味着不能在它们之间共享任何结构体。

康威定律中声称，团队组织结构和微服务结构之间是有直接关联的，所以必须要考虑到客户端与服务端不是由同一个人来维护的可能性，这样会增加对共享结构的维护难度。

在过去的项目中，我们企图在 RESTful 服务与安卓客户端之间共享简单的 Java 类，这导致了无休止的噩梦——运行时故障和周期性的数据损坏。我们应该引以为鉴，避免这种噩梦再次发生。

使用服务捆绑来外部化地址与元数据

上一节中构建了两个服务：一个 catalog 服务和一个 fulfillment 服务。在示例中，catalog 服务能够知道去哪里寻找 fulfillment 服务，因为我们已经直接在 `server.go` 文件中对服务地址进行了硬编码。

这种方式是不可能在生产环境中被使用的，并且对于云环境来说，这种方式也是不友好的。我们真正想要做的就是外部化配置，让应用程序从环境变量当中获取地址与元数据。

回顾一下之前的那些例子就会发现，服务的端口可以被 `PORT` 环境变量重写。这就是外部配置，这个配置可以由 Cloud Foundry 或者 PCF Dev 自动提供。

现在可以通过环境变量去重写这个 fulfillment 服务的地址了。在本例中，应用程序被推送到 Cloud Foundry[1]上安装，并且创建了一个名为 **user-provided service**[2]的服务，这个服务绑定了 catalog 应用程序的地址属性。

如果还没有搭建本地的PCF Dev或在Pivotal Web Services[3]上注册一个免费账户，但还想要继续进行以下工作的话，现在就必须要着手完成以上两件事情了，因为本章内容是使用 Cloud Foundry 服务绑定直接获取服务地址的。

前面已经更新了 catalog 服务中的 `server.go` 文件，让它可以从环境变量中读取 fulfillment 服务的地址。代码清单 6.7 展示了如何从 Cloud Foundry 环境中读取该地址。

代码清单 6.7 server.go

```
package service

import (
        "fmt"

        "github.com/cloudfoundry-community/go-cfenv"
        "github.com/cloudnativego/cf-tools"
        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
```

1 有关 Pivotal Web Services 以及公有的 Cloud Foundry 服务的内容可以在以下链接中查看：http://run.pivotal.io。

2 可以在链接 https://docs. pivotal.io/pivotalcf/devguide/中查看有关 user-provided service 的文档。

3 http://run.pivotal.io/。

```
        "github.com/unrolled/render"
)

// NewServerFromCFEnv decides the URL to use for a webclient
func NewServerFromCFEnv(appEnv *cfenv.App) *negroni.Negroni {
    webClient := fulfillmentWebClient{
        rootURL: "http://localhost:3001/skus"
    }

    val, err := cftools.GetVCAPServiceProperty("backing-fulfill", "url", appEnv)
    if err == nil {
        webClient.rootURL = val
    } else {
        fmt.Printf("Failed to get URL property from bound service: %v\n", err)
    }
    fmt.Printf("Using the following URL for fulfillment backing service: %s\n",
      webClient.rootURL)
    return NewServerFromClient(webClient)
}

// NewServerFromClient configures and returns a Server.
func NewServerFromClient(webClient fulfillmentClient) *negroni.Negroni {
  formatter := render.New(render.Options{
        IndentJSON: true
  })

  n := negroni.Classic()
  mx := mux.NewRouter()

  initRoutes(mx, formatter, webClient)
  n.UseHandler(mx)
  return n
}
func initRoutes(mx *mux.Router, formatter *render.Render, webClient fulfillmentClient) {
  mx.HandleFunc("/", rootHandler(formatter)).Methods("GET")
  mx.HandleFunc("/catalog", getAllCatalogItemsHandler(formatter)).Methods("GET")
  mx.HandleFunc("/catalog/{sku}",
    getCatalogItemDetailsHandler(formatter, webClient)).Methods("GET")
}
```

假设默认使用 `localhost` 作为地址。接下来引用一个开源 `go-cfenv` 包，它可以基于 Cloud Foundry 提供一个抽象的环境，这个环境在 `main.go` 文件中进行初始化。然后构建一个 `cf-tools` 包，这个包中有名为 `GetVCAPServiceProperty` 的方法，该方法可以从指定的服务中查询到指定的属性，并以 Go 语言惯用的模式返回一个结果和一个错误。

通过查询 user-provided 服务的 backing-fulfill 属性，可以得到一个有用的 url 代值，然后可以用这个值替换默认的 fulfillment 服务地址。

可以通过下面的 Cloud Foundry CLI 命令来构建 user-provided service。

```
cf create-user-provided-service backing-fulfill -p "url"
```

这个 CLI 将会提示输入 URL 的值，该 URL 值就是之前推送到 CF 上的应用程序的地址[译注1]。因为所有的 CD pipeline 都以镜像的形式存储在 Docker Hub 中，所以可以很方便地通过下面的简单推送命令来推送应用程序。

```
cf push -o cloudnativego/backing-catalog
```

这个命令之所以可以工作是因为每一个应用程序都包含了一个 manifest.yml 文件，这个文件可以在 GibHub 仓库中找到。举个例子，以下是 fulfillment 服务的 manifest 文件。

```
applications:
- path: .
  memory: 512MB
  instances: 1
  name: backing-fulfillment
  disk_quota: 1024M
```

如果使用 Pivotal Web Services（PWS），则需要更改应用程序的名字，因为该名字已经用来创建了一个路由，而且路由是在所有用户的 PWS 中全局共享的。由于要使用 PWS 来测试这一节，因此设定路由为 backing-fulfillment.cfapps.io.

构建服务并让 fulfillment 和 catalog 应用程序运行在云端后，就可以将服务绑定到 catalog 应用程序上了。

```
cf bind-service backing-catalog backing-fulfill
```

该命令将 backing-fulfill 服务绑定到 backing-catalog 服务上。这个 CF 命令行工具将应用程序名作为第一个参数传给每一个将要绑定它的应用程序。

现在已经创建了一个服务并将它绑定到了 catalog 应用程序上，要重新编排 catalog 应用程序，以便它能感知到新的环境变量。当 catalog 启动时，可以在 CF 产生的应用程序日志中看到以下信息。

```
2016-01-16T16:15:48.96-0500 [APP/0] OUT Using the following URL for
```

译注1　该地址即为前面启动的 backing-fullfillment 应用的路由地址。

```
fulfillment backing service: http://fulfillment.cfapps.io/skus
```

到目前为止都还不错。fulfillment 和 catalog 应用程序都启动并运行在云中，可以尝试访问一下/catalog/，指定任意的SKU值(例如 backing-catalog.cfapps.io/catalog/BOB)。下面的例子将 ***BOB*** 作为 SKU 的值。所有操作都按照预期进行着，观察一下 fulfillment 服务的实时日志，会发现 catalog 服务发送了一个请求给 fulfillment 服务来获取该 SKU 的状态信息（此处更改了一些输出信息防止读者侵入我的个人账号）。

```
$ cf logs fulfillment
Connected, tailing logs for app fulfillment in org (redacted) / space (redacted)
 as (redacted)...
2016-01-16T16:17:53.69-0500 [RTR/0] OUT fulfillment.cfapps.io -
[16/01/2016:21:17:53 +0000] "GET /skus/BOB HTTP/1.1" 200 0 64 "-" "Go-http-client/1.1"
10.10.2.12:39519 x_forwarded_for:"54.84.112.162" x_forwarded_proto:"http" vcap_
request_id:b21bdeee-1403-4cd1-44ad-16e083ea3f37 response_time:0.002252901 app_
id:fc79464e-ab53-42a1-90b0-afbb99f888d7
2016-01-16T16:17:53.69-0500 [APP/0]          OUT [negroni] Started GET /skus/BOB
2016-01-16T16:17:53.69-0500 [APP/0]          OUT [negroni] Completed 200 OK in 115.318µs
2016-01-16T16:18:19.33-0500 [HEALTH/0]       OUT Exit status 0
```

服务发现

到目前为止，本章讨论的服务生态系统都建立在**资源绑定**的概念上。也就是说，当一个服务需要绑定到另一个服务上时，它是怎么知道要访问的另一个服务的地址、凭据以及元数据的呢？

绑定可以通过自动读取配置清单的方式完成，也可以通过持续交付的方式完成，还可以通过人工手动配置的方式来完成。无论使用什么方式完成，配置绑定都必须要明确。

在大多数情况下，这种模式能够很好地工作，人们都比较喜欢亲自控制服务具体的绑定方式。在这种场景下，暴露给应用程序的服务地址是由团队中的某个人来决定的。

另外，有时候我们可能更想得到一个动态的生态服务系统，一个更小且灵活的环境，通过这个环境可以实现运行时服务发现。运行时服务发现可以为检测服务是否正常运行带来好处。

动态服务发现

在明确的绑定当中，不能将应用程序随意地部署到环境中就期望它可以正常运行。固定的绑定意味着它可能会随时失败。如果采用动态服务发现的话，结果就会有所不同。

假如有两个服务：一个游戏服务和一个用户资料服务，游戏服务依赖于用户资料服务。首先，部署用户资料服务到某个环境（如 Cloud Foundry space、Heroku、AWS 等）当中。用户资料服务启动后，会将自己注册到服务发现系统中，然后对外宣称已经准备就绪。

接下来就可以部署游戏服务了。游戏服务也会将自己注册到服务发现系统，并广播自己已经准备就绪。当这个游戏服务企图去访问用户资料服务的时候，它会在注册服务处请求用户资料服务的元数据。它不会从绑定数据中获得服务的地址，而是从注册服务中获得。

通过动态服务发现，像用户资料服务这种后端服务的示例可以被无缝移除，也可以关闭后在另一个地址中重启，亦可以发布多个版本，所有的操作都不需要重新绑定或启动访问用户资料服务的后端服务。

此外，通过注册服务还可以一目了然地查看到哪些服务已启动并正在运行，而无须显式地调用每个服务的运行**状态检查**方法。这是可行的，因为服务与注册服务通信时会向注册服务发送**心跳**，如果心跳停止，不仅消费它的应用程序会感知到，如果该服务在停止心跳后某段时间内没有自行恢复，管理员也会收到通知。

Netflix 的服务发现系统 Eureka

在了解了一个服务发现系统有多了不起之后，大家可能就会想：“要是有一个开源的服务发现系统可以直接拿来使用，那就太棒了！”

恭喜你！Netflix 可以做到。Netflix 已经开源了一整套用来维护其庞大基础设施的软件。Netflix 中有一个名为 *Eureka* 的项目，该项目就是一个服务发现系统。

想要了解关于 Eureka 项目的更多新信息，可以查看其源码，地址是 https://github.com/Netflix/eureka。Eureka 支持服务注册和发现，当然也支持很多其他有用的功能。

Eureka 的入门门槛还是很高的，甚至可能会让人望而却步。我们必须弄清楚如何下载产品（是源代码还是安装程序），然后设置所有的前提条件，这可能与之前在服务器中安装的东西相冲突。还需要花一些时间配置服务器，突然“摆弄”新东西

会带来很多的乐趣。

如果只是想正常启动和运行 Eureka 服务并试用一些基础功能的话，那么最简单的方法就是启动一个 Docker 镜像。之前已经多次提到过 Docker，它是一个非常有价值的工具，使用它可以让复杂的事情在几秒钟内完成。

使用如下命令启动 Eureka 服务。

```
docker run -p 8080:8080 netflixoss/eureka:1.3.1
```

这个-p 参数是用来映射 8080 端口的，将 Docker 容器里面的 8080 端口映射到宿主机的 8080 端口上。Eureka 服务的默认端口就是 8080，所以这个映射是有用的。启动 Eureka 服务之后，我们会收到很多垃圾邮件。过几分钟，当服务的所有启动任务运行完成后，就不会再收到垃圾邮件了，说明服务已经准备就绪。

Docker 宿主机的 IP 地址是 192.168.99.100，当运行测试的时候该 IP 将会作为 Eureka 服务 URL 中的一部分。如果 IP 跟这个不一样，那么就需要更改代码。

现在服务已经运行起来了，我们要用它做些什么呢？如果使用的是 Java，那么可以使用官方支持的客户端库。Netflix OSS 套件中仅有很少的功能特性不是使用 RESTful 模式开发的，因此阅读 Netflix 的详细文档，开发一个自己的客户端也是很方便的。

Eureka 中有一个名为 ***fargo*** 的客户端，可以在以下地址找到这个客户端：https://github.com/hudl/fargo。作者声称该客户端目前还没有完全完成，但是用来对 Eureka 服务进行简单操作已经完全够用了。

在代码清单 6.8 中，我们使用 ***fargo*** 库注册一个应用程序，然后查询 Eureka 看看这个应用程序是否被注册过。因为 Eureka 分离了应用程序与示例的概念，所以可以为同一个应用注册多个示例（这是弹性伸缩云应用的理想选择），Eureka 支持应用程序定期更新状态记录和心跳信息，因此可以用来追踪那些运行缓慢或看起来像是要“挂掉”的示例。

代码清单 6.8 fargo.go

```
package main

import (
    "fmt"

    "github.com/hudl/fargo"
)
```

```
unc main() {
  // For a real app, you'd bind a user-provided service with eureka
  // credentials and URL.
  c := fargo.NewConn("http://192.168.99.100:8080/eureka/v2")

  i := fargo.Instance{
        HostName: "i-6543",
        Port:                 9090,
        App:                  "TESTAPP",
        IPAddr:               "127.0.0.10",
        VipAddress:           "127.0.0.10",
        SecureVipAddress:     "127.0.0.10",
        DataCenterInfo:       fargo.DataCenterInfo{Name: fargo.MyOwn},
        Status:               fargo.UP,
    }

    c.RegisterInstance(&i)
    f, _ := c.GetApps()

    for key, theApp := range f {
        fmt.Println("App:", key, " First Host Name:", theApp.Instances[0].HostName)
  }

  app, _ := c.GetApp("TESTAPP")
  fmt.Printf("%v\n", app)
}
```

使用 NewConn 方法创建一个与 Eureka 之间的新连接，该连接的 IP 地址就是 Docker 宿主机的 IP 地址，端口是启动 Eureka 服务时映射的端口。接下来构建一个应用程序示例并向 Eureka 服务注册。最后测试能够遍历所有应用程序并分别获取应用程序的数据。

当遍历所有已注册服务的时候，一个简单的输出如下。

```
App: TESTAPP First Host Name: i-6543
```

在继续阅读下一章之前，建议大家先看看上面这段代码，然后运用 Eureka 来进行注册和示例查询。

获取 Eureka 的 URL

上面的例子对 Eureka 服务的 URL 进行了硬编码，在实际生产环境中是可能使用这种方式的。如果确定应用程序最终是要部署到 Cloud Foundry 上的，那么就使用绑定服务，正如我们在本章中讲过的，需要将 Eureka 示例绑定到应用程序上，

否则就只能依赖于简单的环境变量了。

读者练习

此次读者练习将会把在本书中学到的一切应用于实践中。强烈建议大家花时间进行此练习，因为这会慢慢形成肌肉记忆，以便以后在更复杂的生态系统中快速有效地构建和部署服务。

本练习将改造 catalog 与 fulfillment 服务，使它们可以被注册到 Eureka 服务中，修改 catalog 服务，让它从 Eureka 注册服务中来获取 fulfillment 服务的 URL。

在这部分练习中，当向 Eureka 中注册服务时可以使用硬编码信息。当然，完成这个练习有很多方法，下面列出了一些比较宏观的推荐步骤。

1. 在 GitHub 中 fork 本章 `backing-catalog` 和 `backing-fulfillment` 存储仓库的副本，这能够方便大家追踪工作进度、与原始仓库版本对比、使工作始终保持在一个正确的 Go 工作区中。

2. 参考 ***fargo*** 库的示例代码，修改 fulfillment 服务的代码，以便在启动时可以自动注册。需要从绑定的 user-provided service 中获取 Eureka 的 URL。

3. 修改 catalog 服务，以便在启动时自动注册。仍然要从绑定的 user-provided service 中获取 Eureka 的 URL。提示：可以将同一服务同时绑定到这两个应用程序上，只要这些应用程序在同一空间内即可。

4. 修改 catalog 服务，使其使用 ***fargo*** 向 Eureka 请求 fulfillment 服务的主机信息。

进阶操作

修改刚刚完成的练习，使其注册的主机信息为非硬编码。修改代码，使用 `cf-env` 库查询应用程序的路由信息，使应用程序在 Cloud Foundry 中启动能够自动注册（在 PCF Dev 和 Pivotal Web Services 环境下运行都可以）。

除 Eureka 服务之外，不需要其他的明确绑定，更新的 catalog 服务就能够发现并连接部署在云中的 fulfillment 服务。

提示

可以在以下地址浏览 cf-env 项目的文档以及测试用例：https://github.com/cloudfoundry-community/go-cfenv。大家将会看到如何使用环境对象去查询一个应

> 用程序的名称和主机名。不要担心端口的问题，因为 Cloud Foundry 将会把所有的应用程序暴露到 80 端口（或者 SSL 的 443 端口）上。

本章小结

在这一章中，我们逐渐开始从微服务的河堤驶向充满无数服务的汪洋大海。本章谈到了一些模式和实践经验，关于何时应该何时不应该在服务之间共享数据，服务的配置和发现，以及服务之间的通信等。

这些都是必要的，因为在后面的章节中我们会发现，微服务从来不独立存在于隔离的环境中。如果真心想学习如何构建既能独立扩展又能融入到健壮生态系统中的云原生服务，就必须弄明白本章中讨论的所有概念。

7 构建数据服务

即使是傻瓜也能写出被计算机理解的代码，而只有优秀的程序员才可以写出人类能读懂的代码。

——Martin Fowler

谈及数据操作的时候，人们会自动联想到某些编程语言，想当然地认为 Java 和 C#基本可以操作任何数据库，如果有连它们都不能操作的数据库，那么就没必要使用该数据库了。

接下来可能有人要一脸严肃地问：“Go 语言可以操作数据库吗？”网上存在着一些关于 Go 语言的谬论，试图说服人们去相信 Go 只适合编写一些命令行程序，并且只在一些特定类型的任务中才有用，应该把“真正的工作”留给其他编程语言。

我们绝对不会同意这种说法，写这本书的一个核心目的是告诉那些对以上言论信以为真的人，这种说法绝对是空穴来风。

在本章中，我们将修改 Go-playing 微服务。在第 5 章中，我们使用了一个基于 match 切片的内存存储库构建了此服务。本章将创建一个基于 MongoDB 的替代存储库。

本章涵盖的内容有以下几方面。

- 在 MongoDB 中构建存储库（test-first）。
- 使用新的存储库来更新 Go 服务。
- 集成数据库的测试。

构建 MongoDB 存储库

本节将介绍扩展小服务示例的整个过程，以便它可以变成真正的无状态。为此，我们将会实现一个与 MongoDB 数据库进行通信的程序。

为什么选择 MongoDB

如果你还不熟悉 MongoDB，请先花几分钟时间去了解它。MongoDB 本质上是一个文档数据库，它可以用来存储和检索 JSON 文档。与传统的关系型数据库不同，MongoDB 的同一个集合中的两个文档不必具有相同的 schema。有的人喜欢它的这种灵活性，有的人则认为这是一种乍看有益但实际上会导致毁灭的技术，因此我们要自己抉择。可以在 https://www.mongodb.org/中查阅关于 MongoDB 的详细文档。

注意，不需要在开发环境中安装 MongoDB，因为我们已经使用了 Wercker 和 Docker。一切所需要的应用都可以从 Docker Hub 镜像仓库中免费获得。

在第 5 章中，我们构建了一个服务，允许客户添加和查询 Go 游戏的单个 match。这个服务是有状态的，同时在内存中维护着一个 match 列表。这意味着如果进程死亡或重新启动，保存在该服务器中的所有当前 match 都将丢失。如果将该进程扩展为多个示例，每个示例都有自己的私有状态，那么后果可能会更糟。

虽然该模型对于学习和实验很有用，但对于实际的应用程序而言是根本不可行的。

更新存储库模型

更新在第 5 章中用于创建 `matchRepository` 接口的模型。然后，传递一个 `inMemoryMatchRepository` 方法给所有的路由处理器。对路由处理器进行单元测试以确保存储模型上的方法能够被正确调用，另外还要单独测试如下的内存中的 match repository 代码。

```
type matchRepository interface {
        addMatch(match gogo.Match) (err error)
        getMatches() []gogo.Match
        getMatch(id string) (match gogo.Match, err error)
}
```

接下来使用 `gogo-engine` 项目中简单的 `Match` 结构体构建一个内存式的存储库。完成这部分操作后，我们将会构建一个该接口的 MongoDB 版本的实现。

> **偶尔成功**
>
> 如果大家的目光足够敏锐，可能会注意到内存式存储库的一些问题：当有新移动时，仅能够保留对象的最新状态，而且这还是偶尔才有效的。之所以这样是因为它在内存中存储指向对象的指针，存储库能自动发现游戏面板的变化。一旦对象进入无状态时，我们就会灵敏地察觉到存储库接口的一个缺陷：它没有明确的 update 方法。一旦编写的测试执行失败并捕获到这个 bug，就应该尽快纠正。

通过 Go 来操作 MongoDB

Go 项目中引用的每一个外部依赖都有非常严格的评估原则。我们需要明确哪些依赖不用自己写，保证使用这个依赖可以解开关键性的难题，这会使工作更容易。许多团队并没有严格执行，因为这在很大程度上是由个人来决定的。

在我看来，如果需要依赖某些依赖，就要对这些依赖负责。无论它的质量是好是坏，都是代码库的一部分。

在这种情况下，我们几乎不会选择自己编写 MongoDB 驱动程序。我们可能更希望扩展或包装已有的依赖包，而不用去重新发明“轮子”——一个复杂的数据库驱动程序。

浏览 MongoDB 的网站，上面列出了目前唯一被官方支持的 Go 驱动程序 `mgo`，其文档可以在 https://godoc.org/labix.org/v2/mgo 中找到。

使用 `mgo` 创建和应用 **session**。使用 `Dial` 函数在 `mgo` 中创建 session，如果继续学习 Go 并探索其他网络库的源代码，就会发现使用 `Dial` 函数是一个相当惯用的模式。

```
session, err := mgo.Dial(url)
```

创建了 session 后，就可以用这个 session 去创建新的文档或查询已存在的文档了，代码如下所示。

```
result := Person{}
err = c.Find(bson.M{"name": "Buckshank"}).Select(bson.M{"phone": 0}).One(&result)
if err != nil {
   panic(err)
}
```

在这个示例中，我们先查询一个集合（用变量 `c` 表示），找到具有要查找的 `name`

属性的文档，找到后筛选出该人的电话号码。

使用 mgo 驱动程序时会有一些奇怪的地方，特别是在云中使用时。例如，在某些情况下，数据库的连接可能会丢失，数据库虚拟机的主机名或 IP 地址可能从某国家的一个海岸漂移到另一个海岸。在这种情况下，需要检测到失效连接并唤醒它。

出于这个原因，我们特别为在 Pivotal 内部使用的需要与 MongoDB 进行通信的 Go 服务开发了一个名为 cfmgo 的包。也可以直接使用 mgo 而不使用这个库。当然，本着开源的精神，大家可以给我们的库贡献代码使它变得更好！

cfmgo 包还有许多用于将 Cloud Foundry 服务绑定到应用程序上的包装器和快捷方式，例如 MongoDB 的商业版本。我们还实现了一个包装模式，它允许暴露一个统一的数据结构，该数据结构在同一个结构体中封装了成功和失败两种查询，可以在 https://github.com/cloudnativego/cfmgo 中找到 cfmgo 的源码及相关文档。

以 Test-First 方式编写 MongoDB 存储库

既然已经为内存式存储库编写了单元测试，那么合适的做法便是编写与内存存储库具有相同断言行为的测试用例，但是要用 MongoDB 替换一下。

在开始编码之前，为艰巨的数据库单元测试任务投入一点时间是值得的。当测试 HTTP 处理器时，我们断言它可以接收、解析并转发数据封装到适当的地方。HTTP 处理器的单元测试不必关心转发的目标（在本书中是指存储库）是否正常工作，因为我们假设存储库已经通过了单元测试。

因此，当对存储库进行单元测试时，我们需要断言各种上层存储库方法已经正确转化成了相应的底层调用。不过，不必测试底层功能是否能够如预期运行，因为数据库驱动程序的作者已经做过了测试（理论上）。

唯一需要关心的端到端的功能测试是**集成**测试，有关集成测试的内容将在本章的后续部分中进行讨论。

测试存储库只是简单地包装相应的底层数据库连接函数的方法，掌握这一点后就可以开始创建单元测试了。

从内存式存储库测试的命名规范和出发点可以猜测到，我们希望对 MongoDB 存储库进行以下测试。

- TestAddMatchShowsUpInMongoRepository。确保向存储库添加 match 时调用适当的底层函数

- `TestGetMatchRetrievesProperMatchFromMongo`。确保可以从存储库中检索单个 match。
- `TestGetNonExistentMatchReturnsError`。确保存储库具有更高级别的错误处理机制，以便在尝试获取不存在的 match 时返回错误。

阅读代码时可以发现，实际上我们不会编写测试来断言是否能在其他文档的集合中查找到 ID 为 *some-id* 的文档。因为这样做就必须完成一个基于 BSON 的查询引擎的完整实现。如果发现为了测试自己的用例而正在重写整个库，这就意味着需要重新评估自己的技术了。

所以此处相较于理想主义，我们选择了实用性。确保在集成测试中应用这个测试用例，但是为了获得那额外 1%的测试覆盖率，必须伪造一个比 MongoDB 库中已经完整测试过的组件更容易出错的软件。

为了构建测试，我们将使用真正的 `cfmgo.Connect` 方法，但是传递一个假的连接器，这个假的连接器会通过稍后构建的 `FakeNewCollectionDialer` 方法来生成。

单元测试代码如代码清单 7.1 所示。

代码清单 7.1 mongorepository_test.go

```
package main

import ("testing"
        "github.com/cloudnativego/cfmgo"
        "github.com/cloudnativego/gogo-engine"
        "github.com/cloudnativego/gogo-service/fakes"

)

var (
        fakeDBURI = "mongodb://fake.uri@addr:port/guid"
)

func TestAddMatchShowsUpInMongoRepository(t *testing.T) {
  var fakeMatches = []matchRecord{}
  var matchesCollection = cfmgo.Connect(
        fakes.FakeNewCollectionDialer(fakeMatches),
        fakeDBURI,
        MatchesCollectionName)

  repo := newMongoMatchRepository(matchesCollection)
```

```
    match := gogo.NewMatch(19, "bob", "alfred")
    err := repo.addMatch(match)
    if err != nil {
        t.Errorf("Error adding match to mongo: %v", err)
    }
    matches, err := repo.getMatches()
    if err != nil {
        t.Errorf("Got an error retrieving matches: %v", err)
    }
    if len(matches) != 1 {
        t.Errorf("Expected matches length to be 1; received %d", len(matches))
    }
}

func TestGetMatchRetrievesProperMatchFromMongo(t *testing.T) {
    fakes.TargetCount = 1
    var fakeMatches = []matchRecord{}
    var matchesCollection = cfmgo.Connect(
        fakes.FakeNewCollectionDialer(fakeMatches),
        fakeDBURI,
        MatchesCollectionName)

    repo := newMongoMatchRepository(matchesCollection)
    match := gogo.NewMatch(19, "bob", "alfred")
    err := repo.addMatch(match)
    if err != nil {
        t.Errorf("Error adding match to mongo: %v", err)

    }

    targetID := match.ID
    foundMatch, err := repo.getMatch(targetID)
    if err != nil {
        t.Errorf("Unable to find match with ID: %v... %s", targetID, err)
    }

    if foundMatch.GridSize != 19 || foundMatch.PlayerBlack != "bob" {
        t.Errorf("Unexpected match results: %v", foundMatch)
    }
}

func TestGetNonExistentMatchReturnsError(t *testing.T) {
    fakes.TargetCount = 0
    var fakeMatches = []matchRecord{}
    var matchesCollection = cfmgo.Connect(
        fakes.FakeNewCollectionDialer(fakeMatches),
        fakeDBURI,
```

```
        MatchesCollectionName)

    repo := newMongoMatchRepository(matchesCollection)

    _, err := repo.getMatch("bad_id")
    if err == nil {
        t.Errorf("Expected getMatch to error with incorrect match details")
    }

    if err.Error() != "Match not found" {
        t.Errorf("Expected 'Match not found' error; received: '%v'", err)
    }
}
```

在每个测试中，我们都创建了一个新的真实的 `cfmgo.Collection` 实例，但是传入了假的集合 dailer，这个 dailer 允许我们为后备存储器提供一些基本的伪造集合。接着要在测试中调用存储库方法，并做一些断言，当相应的底层 MongoDB 函数被调用时，这些断言会提示我们。

代码清单 7.2 中包含了伪造连接器的源码。再次强调，我们要做的不是模拟一个真正的 MongoDB 数据库，而是提供一个假的入口点以测试存储库是否调用了正确的底层方法。例如，`Find` 方法实际上并没有检查任何的搜索条件，如果检查，我们则要在伪造数据中进行功能测试，而不会对真正重要的东西进行测试。

代码清单 7.2 fakes/fake.go

```
package fakes

import (
        "encoding/json"
        "strconv"

        "github.com/cloudnativego/cfmgo"
        "gopkg.in/mgo.v2"
)

var TargetCount int = 1

//FakeNewCollectionDialer -
func FakeNewCollectionDialer(c interface{})
    func (url, dbname, collectionname string)(col cfmgo.Collection, err error) {
    b, err := json.Marshal(c)
    if err != nil {
        panic("Unexpected Error: Unable to marshal fake data.")
```

```
    }

    return func(url, dbname, collectionname string)
        (col cfmgo.Collection, err error) {
          col = &FakeCollection{
                  Data: b
          }
          return
    }
}

//FakeCollection -
type FakeCollection struct {
    mgo.Collection
    Data  []byte
    Error error
}

//Close -
func (s *FakeCollection) Close() {

}

//Wake -
func (s *FakeCollection) Wake() {

}

//Find -- finds all records matching given selector
func (s *FakeCollection) Find(params cfmgo.Params, result interface{})
      (count int, err error) {
    count = TargetCount
    err = json.Unmarshal(s.Data, result)

    return
}

//FindAndModify -
func (s *FakeCollection) FindAndModify(selector interface{}, update interface{},
     result interface{}) (info *mgo.ChangeInfo, err error) {
    return
}

//UpsertID -
func (s *FakeCollection) UpsertID(id interface{}, result interface{})
     (changeInfo *mgo.ChangeInfo, err error) {
     var col []interface{}
```

```
    err = json.Unmarshal(s.Data, &col)
    if err != nil {
          return
    }

  col = append(col, result)
  b, err := json.Marshal(col)
  if err != nil {
        return
  }
  s.Data = b
  changeInfo = &mgo.ChangeInfo{
        Updated: 1,
        Removed: 0,
        UpsertedId: id,
  }
  return changeInfo, nil
}

//FindOne -
func (s *FakeCollection) FindOne(id string, result interface{}) (err error) {
  i, err := strconv.Atoi(id)
  if err != nil {
        return
  }
  var col []interface{}
  err = json.Unmarshal(s.Data, &col)
  if err != nil {
        return
  }
  b, err := json.Marshal(col[i])
  if err != nil {
        return
  }
  err = json.Unmarshal(b, result)
  return
}
```

集成测试一个 Mongo-Backed 服务

集成测试是一件困难的事情。没有简单的按钮，没有快速修复，无论多么有经验，集成测试都需要我们尽心尽力，要求测试人员要富有条理和远见卓识。

然而，这不会成为工作中的阻碍，因为已经有一些工具和技术可以帮助我们克服障碍，使编写集成测试成为开发工作中重要但并不困难的组成部分。

集成临时 MongoDB 数据库

集成测试中最难的部分就是集成本身。在本书的例子中，我们在微服务中利用 MongoDB 持久化来支持 Go 游戏（可以在 https://github.com/cloudnativego/gogo-service 中找到）。此服务需要一个真正的 MongoDB 数据库才能正常工作。那么如何设置临时的 MongoDB 数据库呢？

使用其他语言的传统企业级数据库驱动程序时，人们经常使用同一种类的 JDBC driver 对内存版本的数据库进行集成测试。通常这样是可以正常工作的，直到遇到下面的情况：内存式的存储库工作正常，但是实际的后端存储（例如 Postgres 或 MySQL）并不支持该操作而导致应用程序失败。更糟糕的是，应用程序可能莫名其妙地在生产环境中失败。

真正的集成测试应该使用真实的数据库服务。幸运的是，我们不必在本地工作站中安装数据库服务。我们选择了持续交付工具 Wercker，它可以在自动化构建期间在 Docker 容器中启用后端服务。所有这些后端服务的 Docker 镜像都会通过环境变量连接到运行的应用程序测试镜像上。

这意味着我们可以在每次构建应用程序时使用 Wercker 和 Docker 启动一个空的 MongoDB 数据库，在此期间，可以对服务 API 进行一系列的 RESTful 调用，所有的请求都会流向真正的 MongoDB 数据库。构建完成后，数据库就会消失。我们可以使用 Wercker CLI 在本地运行这个测试，也可以在每次执行 `git commit` 后让这个测试自动运行。

首先，在 `wercker.yml` 文件中添加以下两行代码，指定构建时将依赖 MongoDB 服务。

```
services:
  - mongo
```

当此服务启动后，Wercker 将自动创建几个环境变量，应用程序可以使用这些环境变量与此服务进行通信。Wercker 将把正在进行构建的 Docker 容器链接到运行 MongoDB 服务的 Docker 容器上。

根据此服务的性质，集成测试需要更长的时间来执行，并且需要大量的资源。

我们不希望在集成测试中每次执行 go test 时都要重复进行单元测试。为了方便起见，可以将集成测试放在以下画线开头的目录中，除非明确地指定这个目录，否则它们将被隐藏起来。

现在可以在 wercker.yml 中添加一个集成测试，代码如下。

```
- script:
    name: integration tests
    code: |
      export VCAP_SERVICES=`vcapinate -path=./integrations/vcap.yml`
      export VCAP_APPLICATION={}
      godep go test ./integrations/_test -v -race
```

不必担心 vcapinate 命令。这是我们编写的一个工具，这个工具可以很容易地将一组环境变量（如 Wercker 提供的环境变量）转换为 PaaS 平台提供商（如 Cloud Foundry）可识别的格式。在 Wercker 构建中可以直接使用这个工具，如果想在 Wercker 环境外使用它，则可以在 cf-tools 的 GitHub 仓库中找到，地址是 https://github.com/cloudnativego/cf-tools/tree/master/vcapinate。

以上操作看起来像是增加了不必要的复杂性，但其实并没有那么糟，而且这样做是值得的。这样做的好处是，可以在本地、Docker 容器或云中运行应用程序。

vcap.yml 文件只是为构建 VCAP_SERVICES 环境变量的 vcapinate 工具提供配置信息的。有关服务绑定、环境变量及其在 Cloud Foundry 中的工作原理的更多信息，请查看 https://docs.run.pivotal.io/devguide/deploy-apps/environment-variable.html#VCAP- SERVICES 中的内容。

在下面的代码段中，我们添加了换行符以便读者阅读。如果想得到实际的文件，建议从 GitHub 中下载，而不是直接键入下面的 vcap.yml 文件中的内容。

```
---
userprovided:
- name: mongodb
  credentials:
    url: mongodb://{{.MONGO_PORT_27017_TCP_ADDR}}:{{.MONGO_PORT_27017_TCP_PORT}}/
      some-guid-string
```

当 Wercker 构建正在运行且 MongoDB 服务处于活动状态时，它会提供 MONGO_* 环境变量，如 Wercker 网站上所述，地址是 http://devcenter.wercker.com/docs/services/。请注意，此配置和这些环境变量替换仅用于 Wercker 构建，当将应用程序部署到 Cloud Foundry 上时，这些配置并不是必需的，具体将在下一节中进行介绍。

我们还向 server.go 中添加了一些代码，以便检测 Cloud Foundry 中的 VCAP_SERVICES 环境变量是否存在。如果找到了绑定的 MongoDB 服务，那么就将该服务附加到一个真正的 MongoDB 示例而不是内存式存储库上。以下代码是新添加的 initRepository 函数，可供参考。

```
func initRepository() (repo matchRepository) {
  appEnv, _ := cfenv.Current()
  dbServiceURI, err := cftools.GetVCAPServiceProperty(dbServiceName, "url", appEnv)
  if err != nil || dbServiceURI == "" {
    if err != nil {
      fmt.Printf("\nError retrieving database configuration: %v\n", err)
    }
    fmt.Println("MongoDB was not detected; configuring inMemoryRepository...")
    repo = newInMemoryRepository()
    return
  }
  matchCollection := cfmgo.Connect(cfmgo.NewCollectionDialer, dbServiceURI,
    MatchesCollectionName)
  fmt.Printf("Connecting to MongoDB service: %s...\n", dbServiceName)
  repo = newMongoMatchRepository(matchCollection)
  return
}
```

编写一个集成测试

现在已经完成了一些基础架构，接下来就可以专注于实际的集成测试了。我们的服务公开的 RESTful 方法允许创建新的 match、删除现有 match、查询 match 的详细信息，也可以查询所有 match 的列表。该 API 对于大规模游戏服务来说有点“幼稚”，但它却很适合作为演示用例。

在实际编码之前应该先编写一个测试脚本去测试将要运行的功能，这样做是很有必要的。在本书的例子中，测试将要完成的任务如下。

1. 在启动一个空数据库后向/matches 发出一个 GET 请求，断言将无误地得到一个空数组。

2. 向/matches 发出 POST 请求来添加新 match，断言返回一个 201 HTTP 状态码和一个正确的 JSON 响应。

3. 再次向/matches 发送 GET 请求并断言在响应中会返回一个 match 项。

4. 通过向/matches 发送 POST 请求来添加第二个 match 项，断言会返回一个正确的响应。

5. 向/matches/*{id of first match}*发送一个 GET 请求，断言返回预期的详细信息。

6. 通过向/matches/*{id}*/moves 发送 POST 请求来为第一个 match 项添加一次 move，断言得到一个 201 HTTP 状态码和一个正确的 JSON 响应。

7. 查询第一个与第二个 match 项的详细信息，断言第一个 match 项将被正确地 move，而第二个并不会被 move。

8. 给第二个 match 项添加一次 move。

9. 查询第一个与第二个 match 项的详细信息，断言第二个 match 项将被正确地 move，而第一个不会被 move。

10. 对于实际的应用程序，返回并添加许多负面案例，以证明错误的输入能被正确处理，重复相同的请求可以正常工作等。

可能还有一些其他场景可以运行集成测试，但其中的许多场景测试的是 gogo 游戏引擎本身的风险，而不是服务功能和持久层的性能。

代码清单 7.3 显示了集成测试的大部分代码。我们省略了工具函数的代码，但大家可以像前面一样，在 GitHub 上获得它们。此外，本书中省略了一些原始的 JSON 数据封装，这些也可以在 GitHub 上获得。

代码清单 7.3 integrations/_test/integration_test.go

```
func TestIntegration(t *testing.T) {
  // Get empty match list
  emptyMatches, err := getMatchList(t)
  if len(emptyMatches) > 0 {
    t.Errorf("Expected get match list to return an empty array; received %d",
             len(emptyMatches))
  }
  // Add first match
  matchResponse, err := addMatch(t, firstMatchBody)
  if matchResponse.PlayerBlack != "Hingle McCringleberry" {
   t.Errorf("Didn't get expected black stone player name from creation, got '%s'",
      matchResponse.PlayerBlack)
  }

  matches, err := getMatchList(t)
  if err != nil {
        t.Errorf("Error getting match list, %v", err)

  }
```

```
if len(matches) != 1 {
        t.Errorf("Expected 1 active match, got %d", len(matches))
}
if matches[0].PlayerWhite != "L'Carpetron Dookmarriott" {
        t.Errorf("Player white name was wrong, got %s", matches[0].PlayerWhite)
}
// Add second match
matchResponse, err = addMatch(t, secondMatchBody)
if matchResponse.PlayerBlack != "J'Dinkalage Morgoone" {
   t.Errorf("Didn't get expected black stone player name from creation, got '%s'",
    matchResponse.PlayerBlack)
}

matches, err = getMatchList(t)
if err != nil {
        t.Errorf("Error getting match list, %v", err)
}
if len(matches) != 2 {
        t.Errorf("Expected 2 active match, got %d", len(matches))
}
if matches[1].PlayerWhite != "Devoin Shower-Handel" {
        t.Errorf("Player white name was wrong, got %s", matches[1].PlayerWhite)
}

// Add second match
matchResponse, err = addMatch(t, secondMatchBody)
if matchResponse.PlayerBlack != "J'Dinkalage Morgoone" {
  t.Errorf("Didn't get expected black stone player name from creation, got '%s'",
   matchResponse.PlayerBlack)
}

matches, err = getMatchList(t)
if err != nil {
        t.Errorf("Error getting match list, %v", err)
}
if len(matches) != 2 {
        t.Errorf("Expected 2 active match, got %d", len(matches))
}
if matches[1].PlayerWhite != "Devoin Shower-Handel" {
        t.Errorf("Player white name was wrong, got %s", matches[1].PlayerWhite)
}

// Get match details (first match)
firstMatch, err := getMatchDetails(t, matches[0].ID)
if firstMatch.GridSize != 19 {
      t.Errorf("Expected match gridsize to be 19; received %d",
  firstMatch.GridSize)
}
```

```
    secondMatch := matches[1]

    // Add Move
    addMoveToMatch(t, firstMatch.ID, []byte("(cut for print chapter)"))
    updatedFirstMatch, err := getMatchDetails(t, firstMatch.ID)
    if err != nil {
          t.Errorf("Error getting match details, %v", err)
    }
    if updatedFirstMatch.GameBoard[3][10] != 2 {
          t.Errorf("Expected gameboard position 3,10 to be 2, received: %d",
          updatedFirstMatch.GameBoard[3][10])
    }

    originalSecondMatch, err := getMatchDetails(t, secondMatch.ID)
    if originalSecondMatch.GameBoard[3][10] != 0 {
          t.Errorf("Expected gameboard position 3,10 to be 0, received: %d",
           originalSecondMatch.GameBoard[3][10])
    }

    addMoveToMatch(t, secondMatch.ID, []byte("(cut for print chapter)"))

    updatedFirstMatch, err = getMatchDetails(t, firstMatch.ID)
    if updatedFirstMatch.GameBoard[3][10] != 2 {
          t.Errorf("Expected gameboard position 3,10 to be 2, received: %d",
           updatedFirstMatch.GameBoard[3][10])
    }

    updatedSecondMatch, err := getMatchDetails(t, secondMatch.ID)
    if updatedSecondMatch.GameBoard[3][10] != 1 {
        t.Errorf("Expected gameboard position 3,10 to be 1, received: %d",
         updatedSecondMatch.GameBoard[3][10])
    }
}
```

当该集成测试代码就绪并且修改完 `wercker.yml` 文件后，接下来就可以使用 `buildlocal` 脚本来执行 Wercker 构建和集成测试了。我们已经删掉了大部分与集成测试执行结果无关的输出，具体如下。强烈建议大家从 GitHub 中获取最新的代码，并自己运行一下。

```
$ ./buildlocal
--> Executing pipeline
--> Running step: setup environment
--> Running step: wercker-init
--> Running step: setup-go-workspace
--> Running step: go get
```

```
--> Running step: go build
--> Running step: env
--> Running step: go test
--> Running step: integration tests
Connecting to MongoDB service: mongodb...
=== RUN TestIntegration
[negroni] Started GET /matches
[negroni] Completed 200 OK in 1.655502ms
        Queried Match List OK
[negroni] Started POST /matches
[negroni] Completed 201 Created in 8.447573ms
        Added Match OK
[negroni] Started GET /matches
[negroni] Completed 200 OK in 1.317576ms
        Queried Match List OK
[negroni] Started POST /matches
[negroni] Completed 201 Created in 1.032064ms
        Added Match OK
[negroni] Started GET /matches
[negroni] Completed 200 OK in 1.037217ms
        Queried Match List OK
[negroni] Started GET /matches/0b2b3f3d-d3cb-4ae5-9a05-8a84f3416a98
[negroni] Completed 200 OK in 1.159048ms
        Queried Match Details OK
[negroni] Started POST /matches/0b2b3f3d-d3cb-4ae5-9a05-8a84f3416a98/moves
[negroni] Completed 201 Created in 3.722095ms
        Added Move to Match OK
[negroni] Started GET /matches/0b2b3f3d-d3cb-4ae5-9a05-8a84f3416a98
[negroni] Completed 200 OK in 1.237185ms
        Queried Match Details OK
[negroni] Started GET /matches/c8c76740-70a8-4732-9127-0be77764b797
[negroni] Completed 200 OK in 1.189956ms
        Queried Match Details OK
[negroni] Started POST /matches/c8c76740-70a8-4732-9127-0be77764b797/moves
[negroni] Completed 201 Created in 2.356393ms
        Added Move to Match OK
[negroni] Started GET /matches/0b2b3f3d-d3cb-4ae5-9a05-8a84f3416a98
[negroni] Completed 200 OK in 1.468825ms
        Queried Match Details OK
[negroni] Started GET /matches/c8c76740-70a8-4732-9127-0be77764b797
[negroni] Completed 200 OK in 1.000479ms
        Queried Match Details OK
--- PASS: TestIntegration (0.03s)
PASS
ok      github.com/cloudnativego/gogo-service/integrations/_test 1.045s
--> Running step: copy files to wercker output
--> Steps passed: 35.81s
--> Pipeline finished: 36.92s
```

如果这个测试通过，那么大家就应该对这个服务拥有高度自信了，这个服务会按照预期运行，MongoDB 存储库的包装器会正常工作，所有的细节，如持久性、封包和跨层数据转换等，都会如期正常运行。

当然，实际运行应用程序时，没有能够完全替代真实世界的场景，所以这是我们接下来要努力的方向。

在云中运行

到目前为止，我们已经在 Wercker 中运行了集成测试，这使我们对自己编写的应用程序产生了一定的信心。然而，我们仍然需要在云中部署和运行应用程序，这样就可以通过真实环境来进行测试了。

如果 Cloud Foundry 命令行客户端配置已经指向了 Pivotal Web Services 账户或 MicroPCF 的本地安装环境，我们便可以继续下面的步骤了。

后端服务的配置

在 Pivotal Web Services 中，大家可以浏览 **marketplace** 寻找想要的服务并把它们添加到指定空间中。服务被添加到空间后，可以以手动或提前在 manifest 中声明的方式来绑定应用程序。

图 7.1 展示了 PWS 中的 ***gogo*** 空间，此处配置了一个 MongoDB 实例。

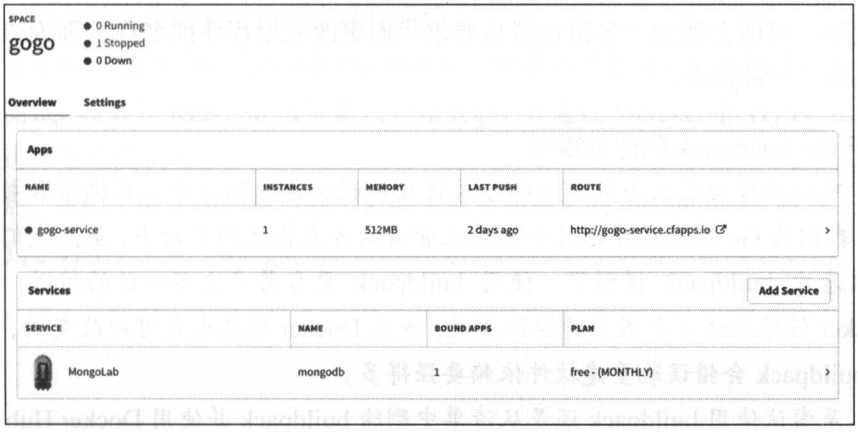

图 7.1 Pivotal Web Services 空间
来源 http://run.pivotal.io

为应用程序设置了空间后，我们可以创建一个 manifest.yml 文件，告诉 Cloud Foundry CLI 如何推送应用程序。编写的应用程序代码会依赖于一个名为 mongodb 的后端服务，这种情况下便可以编写下面的 manifest 来指定这种强制依赖。

```
applications:
- path: .
  memory: 512MB
  instances: 1
  name: gogo-service
  disk_quota: 1024M
  command: gogo-service
  buildpack: https://github.com/cloudfoundry/go-buildpack.git
  services:
   -mongodb
```

如果试图将应用程序推送到 Cloud Foundry 上，而名为 mongodb 的服务不存在，则推送会失败。从实验和文档中可以得知，MongoDB 服务包含了一个名为 url 的属性，它包含主机名、端口、用户名、密码以及私有的 MongoDB 示例中的所有必要连接信息。

此时，我们可以在应用程序的根目录中打开一个终端窗口，并输入以下命令。

```
cf push
```

执行该命令将创建一个名为 gogo-service 的应用程序，该应用程序使用 Go buildpack 命令在云中进行编译，并自动绑定到名为 mongodb 的服务上。请注意，我们已经在 Pivotal Web Services 中使用路由 gogo-service.cfapps.io 创建了一个应用程序，可能会遇到一条错误消息要求我们更改应用程序的名称，因为该应用程序的名称已经被占用。

> **关于 Go buildpack 的注意事项**
>
> 可以选择自己认为最合适的方式将应用程序推送到云中。在编写本书的过程中，我们将 Go 应用程序通过手动建立管道的方式推送到了云中，虽然我们已经不那么喜欢 buildpack 模型了。使用 buildpack 更容易产生不一致的推送，相比用 Docker 镜像推送一个不变的组件来说，虽然 Docker 镜像也是可以改变的，但是总比 buildpack 会错误地重建软件依赖要强得多。
>
> 是尝试使用 buildpack 还是从清单中删除 buildpack 并使用 Docker Hub 中的镜像来推送应用程序，可以根据自己的需求来决定。

现在，可以打开交互式 REST 工具（我们最喜欢用的是 Chrome 的 ***Postman*** 插件）了，发出请求来创建 match、添加 move、删除和查询 match 详情等。大家将看到，即使是重启应用程序，或将其扩展为 *n* 个实例，数据仍会保留在 MongoDB 示例中。

图 7.2 显示了使用 MongoLab dashboard 程序来查看数据库详细信息的方法。只需在 Pivotal Web Services 中单击 MongoDB 服务上的**管理**链接即可访问此 dashboard。如果通过 Docker 镜像在 PCF Dev 上运行本地 MongoDB，则需要在本地示例中指向一个 MongoDB 管理工具。

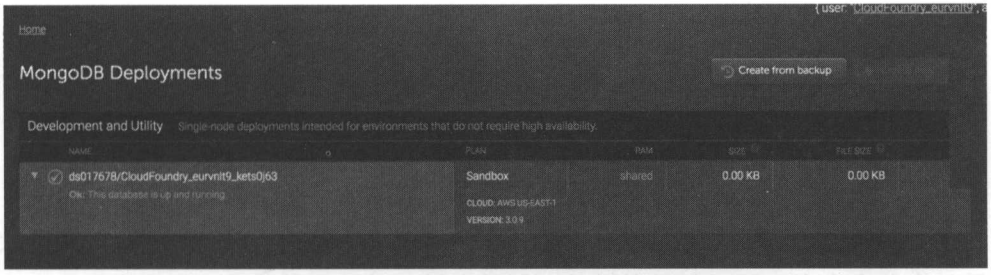

图 7.2 Managed PWS Service 的 MongoLab 仪表盘
来源：PWS (run.pivotal.io)

本章小结

虽然我们都喜欢仅通过 HTTP 进行通信，假定一切都可以通过简单的 REST 调用来实现，但有时还是需要与数据库进行通信。

在本章中，我们了解了如何在 Go 中执行一些基本的任务，例如构建一个数据库的单元测试，将其与 RESTful 服务器代码相关联，并通过 Wercker 和 HTTP 在隔离的环境中进行集成测试等。

最后，我们了解了如何使用基于云的数据库和强大的 marketplaces 服务，并将服务部署到了云中。

8

事件溯源和 CQRS

现实只是一种幻觉，尽管它挥之不去。

——阿尔伯特·爱因斯坦

就像公司里的极客们所想的那样，构建微服务并不都要依赖代码。虽然代码和技术栈肯定是很重要的，但是一些复杂的问题最好还是从架构的层面来解决。

在本章中，我们将讨论如何解决极大规模下的架构设计问题。许多人认为，只要遵守 12 要素法则，就可以无限制地自动扩展云原生应用。我并不同意这个说法。

云原生应用需要能够支持巨大规模的流量。如果希望自己在互联网上部署的应用能够有数以千计甚至数以百万计的人使用，那么就要在架构方面做出相应的调整，只是单纯地扩充应用程序的示例数有时是远远不够的。

本章将讨论两种模式：**事件溯源（Event Sourcing）**模式和 **CQRS**（Command Query Responsibility Segregation，命令查询职责分离）模式，这两种模式都可以解决巨大请求量和吞吐量场景下的应用程序响应问题。本章涉及的主题有以下几个。

- 介绍事件溯源模式。
- 介绍 CQRS 模式。
- 构建高级的 ES+CQRS 代码示例。
  - 构建一个命令处理器服务。
  - 构建一个事件处理器。
  - 构建一个查询处理器服务。

现实源自事件

在与同事和客户常聊起的话题中，没有哪个比事件溯源模式更有趣了。有些人从来没有听说过它，有些人咒骂它，还有一些人使用它工作了很多年竟然还不知道它的名字。

了解事件溯源最简单的方法是类比。就拿大脑来说吧，我们以前可能没有这样想过，人类的大脑其实就是一个事件处理系统，大脑的一部分职责是接收来自五官的刺激。这些事件会进行重新排序（稍后会详细讨论这一点），然后提交给大脑的另一部分，而这部分就是负责构建现实的。

大家是否不停追问过，光的传播速度比声音快，眼睛处理视觉信号所花费的时间却比耳朵处理声音信号所花费的时间要长，那么大脑是如何设法同步声音和视觉来还原现实的？更有趣的是，根据刺激的不同，从接收事件到大脑的认知部分对这些信息做出反应，这个过程可能需要花费几百毫秒的时间。一个典型的例子是，一个被快速踢飞的足球可以在守门员的大脑获取到有重要事情发生的信号之前便绕过守门员直接飞进球门。

实际上，我们认为那就是感官系统在过去收到的信号。无论多么聪明或者警觉，人们所感知的现实都是落后于感官收到的刺激的。所以我们永远都生活在过去，这是不可改变的。

这与构建软件有什么关系呢？你也许感觉很奇怪，其实上面的例子要表达的就是“现实源自事件”这一观点。现实就是一个运行在输入事件流上的函数，如果进一步推想，现实其实就是应用程序当前状态的合集，那么我们就可以明白事件溯源模式不是处理事件输入的某个新潮方案，它其实就是处理事件输入的合适方案。

若以数学方式来阐明，可以声明一个接收事件序列并返回一个状态的函数，如下所示。

```
f(event, ...) = state
```

在传统的状态管理方法中，我们经常维护一些代表所有状态的内存结构。当接收到事件时便根据需要改变状态，而很少关心事件的顺序和状态是如何改变的。这种模型在大流量的场景下会使人陷入困境。

在事件溯源系统中，每收到一个事件，我们都会将其应用到之前算出的状态上，同时产生一个新的状态，如下所示。

$f(state_1, event, ...) = state_2$

不要担心，如果不是很明白这一点，可以在本章的后续内容中更详细地学习这种模式。

事件溯源模式有许多优点，也经常被批评为将事情搞得更“困难”的一种方法。以下几点是一个事件溯源系统的关键原则。

许多人声称事件溯源模式很复杂（或者说彻底失败）的原因之一是他们没有把这些原则作为不可违背的准则。若没有这些准则，事件溯源应用程序就会变得一团糟。

幂等

事件溯源的业务逻辑必须是幂等[1]的。

在真正的事件溯源系统中，`f(e, ...)` 若传入相同的事件序列，那么得到的 `state` 也总是相同的。这不仅仅是事件溯源的副作用，也是事件溯源的一个绝对准则。如果要构建一个事件溯源系统，那么业务逻辑必须是已经确定的。

这也是很多人喜欢像事件溯源模式这样进行函数式编程的原因之一。

当在同一输入流上多次执行已确定的业务逻辑时，所生成的状态必须始终相同。再怎么强调这条准则的重要性也不过分，如果没有这个准则，任何想要完成的操作，包括 CQRS（稍后讨论），都将会失败。如果事件溯源系统失败了，它的影响往往是毁灭性的。

隔离

事件溯源的业务逻辑不能依赖于事件流之外的数据。

对事件溯源系统来讲，有时候一些细节往往会导致事件溯源系统失败，而这样的失败往往是毁灭性的，它们多是由微小、含糊的细节造成的。遇到这些情况的开发者往往就成为事件溯源模式直言不讳的批评者。这就像在不阅读说明书的情况下去组装 IKEA 橱柜，然后组装失败时责怪这个橱柜不好是一样的。

业务逻辑很少在单独隔绝的环境里执行。因为应用程序经常会引用其他数据，更重要的是，应用程序会定期使用外部缓存的数据，即使有些开发人员不这么认为。

---

1 https://en.wikipedia.org/wiki/Idempotence。

这在事件溯源系统中是根本做不到的。来看一个股票交易方面的应用程序范例。假如我们已将其转换为事件溯源系统，并且在股票交易事件流上执行业务逻辑来计算应用程序的状态。一切都看起来很不错，直到应用程序从崩溃中恢复，重复事件流以重建其状态时，我们将会发现交易结果都是错误的，结果我们可能就这样被开除了，没有了工资并丢掉了工作。

这是因为业务逻辑使用了事件流之外的数据来计算结果，比如当前股票的价格。这就意味着在状态计算中使用的股票价格可以在同一事件流的多个定价中发生改变，这违反了事件溯源模式的幂等规则。我们已经说过，幂等是事件溯源模式中最重要并且不可违背的规则。

应用程序处理事件所需的每条信息都必须包含在事件本身中。这个要求虽然很难满足，但没有办法，必须如此，这取决于团队如何设计应用程序。处理这种情况的常见模式是将外部数据变更通知作为事件注入到流中。由于这些变更通知存在于事件流中，并且流是有序的，因此可以通过流来重新运行业务逻辑并获得预期的结果。

> **使用外部时钟将会很糟糕**
>
> 如果应用程序中有这样的业务逻辑：在每天的某个时间做出决策（这在金融应用程序中会经常发生，例如，市场是开盘状态还是休市状态），那么时间不能仅存在于事件流之外。如果时间是外部的，那么测试（或崩溃恢复）将根据不同的时间来执行。这就是为什么几乎每个事件溯源系统都要确保事件必须带有一个时间戳，然而业务逻辑却从不访问外部的“时钟”，因为它存在于事件流之外。如果需要得到关于时间的信息，可以将其存放在计算出的状态中，用来作为最新的当前时间，这个时间可以从接收到的事件中获取。

可测试

如果正确构建事件溯源应用程序，那么该程序将会非常易于测试。

事件溯源应用程序最令我们喜欢的一点是，很多时候测试它们是一件很快乐的事情。它的核心其实就是一个函数，这个函数处理一系列输入，并返回一些状态。此函数必须是幂等的，并且在计算状态时仅限于使用事件流中的信息。

没有比这还要易用且对测试友好的环境了。事实上，我们已经看不到那些丑陋

的测试中令人厌烦的规则和方法了，不再需要复杂的模拟框架或在内存中设置虚假对象来支持流外部的数据，因为测试的函数是幂等且被充分隔离的。

可再现，可恢复

事件溯源应用程序通常需要被优化以支持再现和恢复。

如果大家正在遵循云之道来构建一个事件溯源应用程序，并遵守云原生应用程序的所有准则，那么一定会想知道事件溯源模式是如何满足应用程序无状态需求的。

假设事件流是持久的，那么事件溯源应用程序应该总是可以被写入的，这种情况下当它启动时做的第一件事情就是处理事件流以计算出状态。虽然这样做是合理的，但其实还有更好的选择，我们将在 CQRS 部分进行讨论。

无论是否使用 CQRS，只要有一个持久的事件流，那么应用程序就可以从崩溃中恢复。更重要的是，在启动扩展实例后，它们可以快速地进入稳定状态。

可再现能力的另一个经常被忽视的好处是，它可以用来进行审计与故障排查。如果生产环境中的应用程序出现问题，那么就可以抓取导致问题的事件流，然后在测试环境中运行它，检查应用程序并确定发生了什么问题。现在生产环境中的应用程序很少具有这样的可见性和灵活性。

大数据

事件溯源应用程序通常会生成大量的数据。

相信大家已经猜到，事件溯源系统通常可以产生大量的数据。根据我们拥有的事件数量、事件到达的频率以及每个事件的数据负载大小，可以快速地涉足所谓的“大数据”领域。下一节中我们将列举一些事件溯源的例子，讨论那些可以每天轻松产生数百万个事件，甚至每小时产生数百万个事件的示例。

这个事实本质上既不好也不坏，我们在一开始接触事件溯源时就应该知道。在云和按需分配的基础设施的帮助下，我们绝对可以处理这样的数据量，但需要在设计和架构方面花一些精力，这不是通过简单地“增加容量”就能解决的。

拥抱最终一致性

我们建立的事实是基于事件溯源的，实际上大脑感知的事实是几毫秒前发生的

事情。在大脑接收的事件形成事实之前，大脑需要感知到它们并对它们进行排序。总之，现实是**最终一致**的。

有一个用来解释最终一致性的经典场景是这样的：想象一下，在手机上使用社交媒体软件给别人的帖子添加一条评论。当这个人在浏览器中查看自己的帖子时，他会收到了一条通知，告诉他有人添加了一条评论，但是当他想去看看这条幽默的评论时却发现评论不见了！评论似乎已经消失了。

好不容易撰写了世界最佳网络评论，还没被人注意到就消失不见了，这种感觉近乎崩溃！但是，正当近乎绝望的时候，其他朋友刷新了浏览器，看到了这条评论！

为什么会这样呢？这就是最终一致性的结果。为了保证评论最终会在所有地方显示，该服务的架构师做出了有利于服务可靠性和全球大规模瞬时流量下服务可用性的设计决策。在许多情况下，这是该类问题的最佳解决方案。

事情其实并没有我们想象中那样糟糕，只需等待 2 秒，不可思议的评论就可以从我们的手机中到达朋友的浏览器上。

当应用程序被设计用来处理事件流时，其经常被设计成“即发即弃”的模式。事件从不同的源中异步得到，然后可能由一个完全独立的组件异步处理。这种解耦的方式允许事件接收器和事件处理器分开扩展，它还允许接收器、日志记录器和处理器都具有它们各自的可靠性模型。这是超大规模流量下的微服务模型。

几秒之后，当朋友的社交媒体推送自动更新时，可能会看到因入站事件流造成的更改，这可能需要花点时间。这是完全可以接受的，因为此架构允许基础设施具有基于地理位置优化的事件接收器，或者部署边缘服务以优化移动设备的响应时间。如果移动设备非要一直等待，直到接收到“帖子已被所有朋友收到”的确认信息，那么整个系统将会变得非常慢，甚至完全不可用。

不是所有的事件溯源应用程序都需要实现异步、松耦合、最终一致性。但是，越早接受最终一致性的概念，掌握将事件源与事件溯源模型相结合的方法以及它支持的大规模流量类型，就越容易在云中构建这些应用程序。

CQRS 简介

CQRS（命令查询责任分离）是一种奇特的模式，表示解耦系统的输入和输出。在典型的单例应用程序中，一般同时拥有数据库的写入端点和读取端点。两者都操

作相同的数据库，并且通常在数据库接收到确认或事务提交消息之前，不会收到写入端点的答复。

正如截止到目前所讨论的，这种类型的应用程序在许多情况下仍然有效，但在本章讨论的场景中，这样做是站不住脚的。在大规模、高吞吐量和需要对事件进行复杂处理的情况下，需要运行缓慢的读取查询，每当有新输入的事件进来时，停下来等待它处理完成，这些都是无法容忍的。大家一定也不希望系统中高负载的组件影响健康组件的运行效率。

如图 8.1 所示，这个示意图说明了如何建立一个命令和查询责任分离的事件溯源系统。

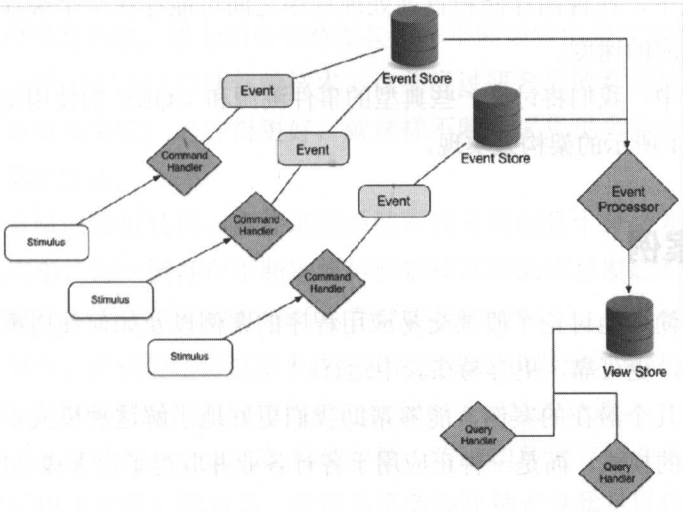

图 8.1 事件源和 CQRS 模式中的事件流

我们将接收到的刺激称为 **command**，将读取操作称为 **query**。从图 8.1 中可以清楚地看到，在该应用的 command 和 query 之间没有直接的依赖关系。事实上，可以据此推断，这两个功能完全可以由不同的服务实现。

在图 8.1 中，从收到刺激信号到执行查询的流程如下。

1. 一些外部的刺激调用命令处理程序。这些刺激可能来自远程传感器的数据，或者是按下移动应用按钮后的响应事件，也可能来自其他操作。

2. 命令处理程序负责创建事件。事件通常按照惯例（或根据其他更好的原因）被命名为以过去时态出现的名词。例如，`StockPriceChangedEvent`、

`GpsCoordinateChangedEvent` 或 `AlarmTriggeredEvent` 等。

3. 此事件被存储在**事件存储器**中。在事件被创建和被存储之间可以有许多的中间件。事实上，该模式的大多数生产级实现都可能包含**消息队列**，用来确保创建的事件都能被正确地存储。

4. 事件处理器可以响应接收的入站事件，并且进行必要的聚合或计算，以便创建发往查询处理程序的新数据。这些数据本身就是为特定查询量身定制的，本质上是一个通过事件计算出的现实，可用于具体的查询。这不是临时缓存，而是持久化的视图或存储库。

5. 当请求进入查询处理程序时，它们会向视图存储库发起极其快速和简单的请求。许多情况下，在视图存储和查询处理程序之间可能存在一个缓存层，用来进一步加快查询响应的速度。

在下一节中，我们将讨论一些典型的事件溯源和 CQRS 的使用案例，这些例子可以使用图 8.1 所示的架构来实现。

事件溯源案例

我们已经简单地讨论了股票交易应用程序的案例以及如何使用事件溯源使该程序更易于测试，更可靠，更容易在云中运行。

下面列举几个潜在的案例，能够帮助我们更好地了解这种模式不是只有几个人使用的很狭隘的模式，而是一种正应用于各行各业并取得了巨大成功的模式。

天气监测

假设我们在全国各地都架设了监控设备，它们被大规模地部署到山顶、海平面、都市和乡村。

这样我们会得到一个包括气压、温度、风速、风向和湿度在内的变化的事件流。每个设备每秒都可以发送多个数据点。因此，每秒会有几十甚至几十万条数据进入事件流中。

这是一个典型的例子，可以说明使用 CQRS 不仅从性能的角度来看是有意义的，而且从设计的角度来看也是如此。天气监测服务的消费者不需要访问底层所有时刻的详细数据，只想知道所有给定传感器的当前值。如果需要深入了解详细数据和历

史数据也可以，但需要优化获得当前天气情况的查询。

此外，该系统可以在后台运行复杂的气象算法，用于满足更高级别的查询，这些查询可能包含有关检测潜在风暴形成的聚合条件或报警条件的信息。

无论大家是否喜欢云原生，都不可能在没有事件溯源和 CQRS 的情况下构建出这样的系统。

互联网汽车

新款的特斯拉电动汽车支持自动驾驶的功能，它利用各种传感器收集周围环境的信息，以辅助驾驶员进行驾驶，实现基本需求，如使汽车保持在车道内，与前车保持适当的距离等。这款汽车上甚至还具有一架相机，可以读取路旁的限速标志。

这是一个**自学习**系统。路上的所有特斯拉汽车不断地将信息发送给特斯拉公司，以帮助他们深入学习和改进自动驾驶技术。这些经过研究后的数据被发回到每一辆电动汽车中，可以帮助它们运行得更好，就这样不断地采集更多的数据进行学习，然后提高自动驾驶性能。

对于这种量级的数据处理，事件溯源系统不仅有用而且十分必要。全球数十万辆特斯拉电动汽车，每一辆都在不断地将传感器捕获到的信息发送给特斯拉公司。特斯拉公司需要在不丢失任何数据或耗尽存储容量的情况下，捕获这数十万个事件流中的每一个事件，并利用这些数据来执行极其复杂的机器学习算法，为 Tesla 移动 App 提供查询如机动车的最近已知位置和当前状态等信息的功能。

试想一下，如果使用一个老式的整体存储，将一切数据存储在内存或传统数据库中而无法进行快速水平扩展的话，仅仅是事务吞吐量本身就足以搞垮一个没有事件溯源、CQRS 和最终一致性的 Web 应用程序。

社交媒体消息处理

很显然，大家可以将社交媒体消息视为事件流。社交活动，例如新信息、新评论、点赞、好友请求以及请求被接受或拒绝等，这些都是事件流当中的事件。

事件溯源应用程序可以消费这些事件，记录需要被记录的数据，并对这些事件流进行处理，使得经过聚合或计算后的信息可以用于查询。也可以使用此类系统来判断特定主题标签的情感，或根据活动流生成建议或预测。

代码示例：管理无人机舰队

在学习了大量的理论概念后，下面我们将编写一个用来管理无人机舰队的应用程序。

在这个应用程序中，命令处理程序接收命令并生成事件，如每个无人机的位置、速度变化和剩余电量信息等。这些事件将被存储到事件存储器中，然后由无人机的事件处理器进行异步处理。最后，事件处理的输出将存储在视图存储器中，供查询处理程序使用，这样应用程序客户端就可以获取无人机舰队中任何无人机的当前状态了。

为了使本书更易阅读，大家可以想象一下，我们可能还需要一个可以接收更改无人机当前目标命令的命令处理程序，虽然我们不会真的实现这些功能。这个处理程序将创建一个 `TargetChangedEvent` 事件。然后，事件处理器将对该事件做出反应并直接向无人机下达命令，通知它改变目标，而不是在查询视图中创建任何信息。

对于一些数据，如无人机的位置等，可以在视图存储中只存储最后接收到的位置信息。其他的计算也可以异步完成，例如计算每个无人机剩余飞行时间的分钟数。可以简单地计算一下，也可以使用像特斯拉公司使用的那种算法，基于最后 n 分钟内消耗的电量信息（从剩余电池的最后几份报告中获得）来评估剩余的电量。

正是像这样的计算和异步处理，使得 CQRS 和事件溯源模式非常吸引人，并且使我们拥有按需动态扩展架构中任何部分的能力。如果还需要添加一些额外的，以前没有做过的重要计算，只需要将其他微服务放入生态系统中，然后读取入站事件流即可。

在本章的下面几节中，我们将编写一个命令处理程序、一个事件处理程序、一个查询处理程序。这些程序都是松耦合且可以独立扩展的。虽然我们不会在示例的事件处理器中进行复杂的聚合和计算，但还是希望大家在创建更复杂的项目时能够以此示例为模板。希望大家真心想用基于 Go 语言编写的微服务去处理所有问题，并将其部署到云端。

在本书的最后，我们为大家提供了一个使用事件溯源和 CQRS 模式的复杂示例。

构建命令处理程序服务

这一节将构建事件溯源系统的三个主要组件中的第一部分——**命令处理**程序服务。此服务的作用是接收传入的命令，将它们转换为事件并放置在队列中，以便事件处理器进行异步处理。

这次不会像通常那样一开始就编写一个失败测试，让我们先来熟悉一下RabbitMQ。由于涉及消息传递中间件的编码在本书中还是第一次出现，所以需要花费几分钟的时间编写一个规范的“hello world”例子，熟悉一下这项技术。

RabbitMQ 介绍

RabbitMQ 是一个实现了高级消息队列协议（AMQP）的开源消息代理服务器。之所以选择 RabbitMQ 是因为它足够简单，启动时间极短，对云的亲和力好。我们是 OTP（开放电信平台）和 Erlang 这两种技术的狂热粉丝，而 RabbitMQ 恰好使用到了这两种技术。

此处介绍的一切都可以在 GitHub 库中找到，地址是 https://github.com/cloudnativego/rabbit-hello。在这个简单的例子中，我们将使用 Docker 启动一个 RabbitMQ 服务器，然后通过一个 Go 应用程序将消息放进队列中，并使用另一个应用程序去接收它。如果使用 Pivotal Web Services（简称 PWS）来进行云部署，则会在 marketplace 上看到一个 RabbitMQ 的标志。

使用以下命令启动 RabbitMQ 服务器，确保端口正确地暴露在了 Docker 宿主机的 IP 地址上（为了方便打印，所有命令都写在同一行中）。

```
docker run -d --hostname my-rabbit --name some-rabbit -p 8080:15672
-p 4369:4369 -p 5672:5672 rabbitmq:3-management
```

请注意，我们使用的标签为 `3-management` 的镜像。这个镜像不仅会启动一个 RabbitMQ 服务，而且还会把内部的管理控制台端口 15672（我们已经将其简单地映射到了常用的 8080 端口上，绑定到 5 位数的端口号在 PWS 中是不允许的）暴露出来。

接下来，我们将使用 RabbitMQ 文档教程中的 `send.go` 文件向队列中添加一条简单的消息，如代码清单 8.1 所示。

代码清单 8.1 send.go

```
package main

import (
        "fmt"
        "log"

        "github.com/streadway/amqp"

)

func failOnError(err error, msg string) {
        if err != nil {
                log.Fatalf("%s: %s", msg, err)
                panic(fmt.Sprintf("%s: %s", msg, err))
        }
}

func main() {
        conn, err := amqp.Dial("amqp://guest:guest@192.168.99.100:5672/")
        failOnError(err, "Failed to connect to RabbitMQ")
        defer conn.Close()

        ch, err := conn.Channel()
        failOnError(err, "Failed to open a channel")
        defer ch.Close()
        q, err := ch.QueueDeclare(
                "hello",    // name
                false,      // durable
                false,      // delete when unused
                false,      // exclusive
                false,      // no-wait
                nil,        // arguments
        )
        failOnError(err, "Failed to declare a queue")

        body := "hello"
        err = ch.Publish(
                "",     // exchange
               q.Name,  // routing key
               false,   // mandatory
               false,   // immediate
               amqp.Publishing{
                       ContentType: "text/plain",
                       Body:        []byte(body),
               })
        log.Printf(" [x] Sent %s", body)
```

```
        failOnError(err, "Failed to publish a message")
}
```

现在，如果大家还纠结一些基础概念，我们强烈建议各位用几杯咖啡的时间来浏览一下 RabbitMQ 的教程并熟悉基本概念。使用此库发送消息的流程如下。

1. 通过 `Dial` 函数连接 AMQP 服务器。
2. 从 `Connaction` 中获取一个 `Channel`。
3. 声明一个 `Queue` 并通过 channel 来设置它的参数。
4. 构建消息体。
5. 通过 channel 发布消息。

请注意，在 channel 上发布消息时仅仅指定了队列的名称。这是以一种假设队列已经创建好的方式在编写代码。强烈建议不要这样做。安全防御性编程是不需要假设任何事情的，这才是最好的方式，特别是在后端服务运行在云端的情况下。

RabbitMQ 在后台启动后（需要修改代码示例中的 IP 地址以匹配 Docker 环境），可以运行如下示例。

```
$ go run send.go
2016/02/08 07:13:10 [x] Sent hello
```

此时，在 RabbitMQ 的队列中会有一条消息：如果停止服务，消息就会丢失。不要担心，RabbitMQ 有许多选项可用于配置队列和消息，以便根据需求来设置持久性和可恢复性。**交换器**（exchanges）是 RabbitMQ 的一个强大功能，它可以将消息一次性传递到多个队列中，还有一些其他功能超出了本书所讨论的范畴。

接收消息与发送消息非常相似，开始差不多一样，包括建立连接、获得 channel 和声明队列等。区别在于，此队列将用于接收消息，而不是发送。

```
msgs, err := ch.Consume(
                q.Name,// queue
                "",    // consumer
                true,  // auto-ack
                false, // exclusive
                false, // no-local
                false, // no-wait
                nil,   // args
        )
        failOnError(err, "Failed to register a consumer")

        forever := make(chan bool)
```

```
        go func() {
                for d := range msgs {
                        log.Printf("Received a message: %s", d.Body)
                }
        }()

log.Printf(" [*] Waiting for messages. To exit press CTRL+C")
<-forever
```

我们使用 Consume 方法来替代 Publish 方法。然后创建一个简单的 **goroutine**（https://golang.org/doc/effective_go.html#goroutines）并让它永远运行下去，始终等待消息的到来，只要消息到来就输出消息体。

```
$ go run receive.go
2016/02/08 07:13:37 [*] Waiting for messages. To exit press CTRL+C
2016/02/08 07:13:37 Received a message: hello
```

因为队列中已经有了一条消息，所以一启动接收器就立即收到了一条消息。如果打开管理控制台（http://*your docker IP*:8080）并单击 **Queues**，可以看到刚创建的 ***hello*** 队列，如图 8.2 所示。

图 8.2 RabbitMQ Admin Console——Queues

来源：http://192.168.99.100:8080/

构建命令处理器服务

命令处理程序的职责是接收传入的命令，将它们转换为与端点无关的事件格式，然后将它们提交到相应的队列中。在这之后，命令处理程序将结束这个请求的任务，并继续处理下一个请求。

正是这种细粒度的责任划分，使这种模式变得可扩展。这也是人们总抱怨它的一个原因：必须创建一些服务，才能完成一个简单的“hello world”示例。当然，我们建立的是基于现实情况的大流量场景下的服务，所以抱怨它让“hello world”变得复杂就有点可笑了。

可以在 https://github.com/cloudnativego/drones-cmds 中找到命令处理程序的完整源代码。由于这个示例比较复杂，所以我们不会花费太多时间来说明每一行代码和所有的测试，我们仅展示最重要的部分，大家可以自己去 GitHub 上研究源码。

代码清单 8.2 展示了一个命令处理程序的示例，项目中的所有命令处理程序都遵循此基本格式。

代码清单 8.2 Command Handlers

```
func addTelemetryHandler(formatter *render.Render,
  dispatcher queueDispatcher) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
        payload, _ := ioutil.ReadAll(req.Body)
        var newTelemetryCommand telemetryCommand
        err := json.Unmarshal(payload, &newTelemetryCommand)
        if err != nil {
              formatter.Text(w, http.StatusBadRequest,
                    "Failed to parse add telemetry command.")
              Return
        }
        if !newTelemetryCommand.isValid() {
              formatter.Text(w, http.StatusBadRequest,
                    "Invalid telemetry command.")

              return

        }

        evt := dronescommon.TelemetryUpdatedEvent{
              DroneID:          newTelemetryCommand.DroneID,
              RemainingBattery:newTelemetryCommand.RemainingBattery,
              Uptime:           newTelemetryCommand.Uptime,
              CoreTemp:         newTelemetryCommand.CoreTemp,
```

```
                ReceivedOn:      time.Now().UnixNano(),
        }
        dispatcher.DispatchMessage(evt)
        formatter.JSON(w, http.StatusCreated, evt)
    }
}
```

接收到命令后要做的第一件事就是将该命令从请求主体中取出来，并将其解析成一个可用的结构体。代码清单 8.2 中生成了一个 `telemetryCommand` 结构体，它的结构如下。

```
type telemetryCommand struct {
        DroneID string 'json:"drone_id"'
        RemainingBattery int 'json:"battery"'
        Uptime int 'json:"uptime"'
        CoreTemp int 'json:"core_temp"'
}
```

如果这个结构体能够被正确解析的话，那么接下来便要调用 telemetryCommand 上绑定的 `isValid` 方法。这种方式很清晰，能确保该命令具有所有的必需字段且格式正确。每个命令类型都有一个相应的 `isValid` 方法（在 `service/types.go` 文件中可以看到）。

接下来，使用传入的命令创建一个 `TelemetryUpdatedEvent`。记住，在命令处理程序中不应该进行任何处理，所以只是简单地添加了一个时间戳而已（参见本章前面介绍的给事件添加时间戳的重要性的内容）。如果有其他外部数据（例如世界统治计划的现状、剩余资金等）需要引入，我们将在命令处理程序中为事件增加这些信息。

最后，命令处理程序的后期职责是将新创建的事件分派到相应的队列中。这是通过调用 `dispatcher.DispatchMessage（evt）` 来完成的。每个处理程序将在其自己的调度程序中传递，因此请求处理程序不必关心如何调度以及往哪里调度，这也使得处理器程序的单元测试变得更容易编写，因为我们可以轻松传递假调度器。

大功告成！如果一切都被正确设置的话，那么这个服务将通过 HTTP POST 接收命令，然后将事件放进 RabbitMQ 队列中。

如果查看 `drones-cmds` 项目的 `manifest.yml` 文件，将看到该项目依赖 `rabbit` 服务。我们已经在 marketplace（可以使用 PCF Dev、Pivotal Web Service 或 PCF 企业版）上创建了一个用来访问 RabbitMQ 的服务示例，并将其命名为 ***rabbit***。

在继续讨论事件处理器之前，可能还需要浏览一遍 GitHub 中的代码。集成测试这一部分会比较有意思，它通过 HTTP POST 提交一系列命令，然后监控 Wercker 中的 RabbitMQ 示例的事件，断言预期的事件会出现在它们各自的队列中 。

与创建的其他所有示例一样，该服务由 Wercker 自动构建并部署到 Docker Hub 上（也可以使用 `docker run` 命令在本地运行），可以将生成的 Docker 镜像部署到云中。

构建事件处理器

事件处理器的职责正如它的名字一样显而易见：处理事件。该服务启动后将监听这三个队列是否正在使用中。每获得一个事件，事件处理器就会去处理它，然后将事件存储在事件存储器中。

在更复杂的实际应用中，事件处理器将基于时间片、聚合或分组进行计算。它也可以执行两种不同类型的写入：一个写入到事件存储器中，将事件归档；另一个写入到视图存储器中以支持查询处理器服务（经常将其称为现实或现实服务）。

尽量让该示例简单易读，以便可以看到我们如何将所有的事件溯源和 CQRS 组合在一起。也可以将此代码作为构建自己的 ES/CQRS 模式应用程序的模板。为了保证简单，我们不会进行任何聚合操作，仅仅是写一个专门提供查询服务的存储器，大家可以看到该解决方案中的所有不同点，并轻松地添加和扩展此功能。

事件处理器的所有代码都可以在 GitHub 上找到，地址为 https://github.com/cloudnativego/drones-events。

关于事件处理器的很有趣的一点是，它没有公共 API。它运行时不需要任何外部的 RESTful 调用，只需要监听事件是否到达了队列即可。

因为我们将会把这个服务部署到云中，所以希望它一直运行，并且能够在微服务生态系统中发挥出色。现在可以选择将服务作为**任务**来运行（没有 HTTP 端点），这里只是告诉平台不要进行健康检查，或者也可以建立一个简单的 HTTP 端点，以便进行本地测试并增加平台的灵活性。

在这个例子中，我们只是在根（/）目录资源上暴露了一些基本文本，这样能够规避那些不支持任务的云环境，并绕过 HTTP 健康检查。

来看一下代码清单 8.3 中的 `NewServer` 函数。

代码清单 8.3 NewServer() for Event Processor

```
// NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {
  formatter := render.New(render.Options{
       IndentJSON: true
  })

  n := negroni.Classic()
  mx := mux.NewRouter()

  initRoutes(mx, formatter)

  n.UseHandler(mx)

  alertChannel := make(chan dronescommon.AlertSignalledEvent)
  telemetryChannel := make(chan dronescommon.TelemetryUpdatedEvent)
  positionChannel := make(chan dronescommon.PositionChangedEvent)

  repo := initRepository()
  dequeueEvents(alertChannel, telemetryChannel, positionChannel)
  consumeEvents(alertChannel, telemetryChannel, positionChannel, repo)
  return n
}
```

以上代码看起来很像之前编写的其他 `NewServer` 函数，然而却多了一些有趣的地方。

首先，该函数创建了三个 Go **channel**。Go channel 允许进行协调异步处理，而不需要担心在使用其他编程语言的多线程时遇到麻烦。有关 Go channel 的更多信息，请参阅 Go 语言的相关资料。

把这些 channel 作为一种管道，将事件从 AMQP（RabbitMQ）队列中取出，并将它们放进这些 channel 中。事件进入这些 channel 后，使用事件处理器来消费它们，最终将最近的事件存储到查询存储器（一个 MongoDB 示例）中。

大家可能已经猜到了，这个服务的绝大多数工作都在 `dequeueEvents` 和 `consumeEvents` 函数中。

代码清单 8.4 显示了 `dequeueEvents` 函数的具体实现。

代码清单 8.4 dequeueEvents()

```
func dequeueEvents(alertChannel chan common.AlertSignalledEvent,
                   telemetryChannel chan common.TelemetryUpdatedEvent,
```

```
                positionChannel chan common.PositionChangedEvent) {
  fmt.Printf("Starting AMQP queue de-serializer...")
  appEnv, _ := cfenv.Current()
  amqpURI, err := cftools.GetVCAPServiceProperty("rabbit", "url", appEnv)
  if err != nil {
        fmt.Println("No Rabbit/AMQP connection details supplied.
ABORTING. No events will be dequeued!!!")
        return
  }
  fmt.Printf("dialing %s\n", amqpURI)
  conn, err := amqp.Dial(amqpURI)
  if err != nil {
        fmt.Printf("Failed to connect to rabbit, %v\n", err)
  }
  ch, err := conn.Channel()
  if err != nil {
        fmt.Printf("Failed to open AMQP channel %v\n", err)
  }

  alertsQ, _ := ch.QueueDeclare(
        alertsQueueName, false, false, false, false, nil,
  )

  positionsQ, _ := ch.QueueDeclare(
        positionsQueueName, false, false, false, false, nil,
  )

  telemetryQ, _ := ch.QueueDeclare(
        telemetryQueueName, false, false, false, false, nil,
  )

  alertsIn, _ := ch.Consume(
        alertsQ.Name, "", true, false, false, false, nil,
  )
  positionsIn, _ := ch.Consume(
        positionsQ.Name, "", true, false, false, false, nil)

  telemetryIn, _ := ch.Consume(
  telemetryQ.Name, "", true, false, false, false, nil,
  )

  go func() {
        for {
          select {
           case alertRaw := <-alertsIn:
              dispatchAlert(alertRaw, alertChannel)
           case telemetryRaw := <-telemetryIn:
              dispatchTelemetry(telemetryRaw, telemetryChannel)
```

```
            case positionRaw := <-positionsIn:
                dispatchPosition(positionRaw, positionChannel)
            }
        }
    }()
}
```

此代码的基本流程是声明队列，从队列中消费事件，然后在一个无限循环中使用 Go `select` 语句将消息从队列中取出并分派到相应的 channel 上。在分派动作中需要将原始 AMQP 消息转换成事件结构体，如下面的 `dispatchAlert` 函数所示。

```
func dispatchAlert(alertRaw amqp.Delivery, out chan common.AlertSignalledEvent) {
  var event common.AlertSignalledEvent
  err := json.Unmarshal(alertRaw.Body, &event)
  if err == nil {
        out <- event
  } else {
        fmt.Printf("Failed to de-serialize raw alert from queue, %v\n", err)
  }
  Return
}
```

成功地将原始的 AMQP 格式的数据转换成强类型的事件结构体后，就可以将事件发送到 channel 内了，那里有 goroutine 正在等待，这些都已经在 `consumeEvents` 函数中定义过了，如代码清单 8.5 所示。

代码清单 8.5 consumeEvents()

```
func consumeEvents(alertChannel chan common.AlertSignalledEvent,
 telemetryChannel chan common.TelemetryUpdatedEvent,
 positionChannel chan common.PositionChangedEvent, repo eventRepository) {
 go func() {
       fmt.Println("Started event consumer goroutine")
       for {
              select {
               case alert := <-alertChannel:
                      processAlert(repo, alert)
               case telemetry := <-telemetryChannel:
                      processTelemetry(repo, telemetry)
               case position := <-positionChannel:
                      processPosition(repo, position)
               }
       }
 }()
}
```

这是一个非常简单的函数。它等待新传入的消息，使用 select 语句进行分发，并分别处理它们。可以在这里插入额外的处理程序，其复杂度取决于业务逻辑处理。该示例实现的处理只是将最近接收到的事件“upserting”（更新并插入）到 MongoDB 存储库中。

在更高级和适当的拆分示例（如本书最后的示例）中，实际上可能会将计算出来的状态提交到微服务 *reality* 中，而不是直接与 MongoDB 数据库进行通信。这样能够让事件存储器和现实/查询存储器在事件处理器之外进行扩展。

这个示例的逻辑显示在 processAlert 函数中。

```
func processAlert(repo eventRepository, alertEvent common.AlertSignalledEvent) {
  fmt.Printf("Processing alert %+v\n", alertEvent)

  repo.UpdateLastAlertEvent(alertEvent)
}
```

以这种方式处理事件有一个巨大的优势：如果有哪个事件处理器示例停顿了，可以简单地扩展出多个事件处理器，因为工作负载是由队列分派的，也可以通过设置 RabbitMQ 服务器以保证同一个消息不会发送给两个不同的消费者。

来看一下事件处理器服务的 manifest.yml 文件，它依赖如下两个后端服务。

- rabbit
- mongoevent-rollup

请注意，事件处理器使用的 RabbitMQ 服务与命令处理程序使用的 RabbitMQ 服务必须是同一个。如果不是，那么将永远也不会看到命令处理程序产生的消息，因为它们将分别被存储在孤立隔绝的 Rabbit 示例中。

展望未来，我们知道事件处理器和查询处理程序将共享同一个 MongoDB 服务（mongoevent-rollup），以便查询服务能够查看到最近被存储和处理的事件。

违反模式的警告

事件处理器和查询处理器都与同一个 MongoDB 数据库通信，这样做违反了微服务的关键原则：永远不要将数据库作为集成层。这样做只是为了降低示例的复杂度，在实际的应用程序中，事件处理器会将事件写入事件存储器或经过计算后的状态中（Reality 服务）。

还需要指出的是，这是微服务开发人员之间的争论焦点。一些人认为两个服

> 务共享同一个数据库是好的，其他人（像我们）则认为这是不可接受的。具体还要由我们自己决定。

对事件处理器进行集成测试

对事件处理器服务进行集成测试的主要目的是，确保它可以正确处理传入的事件并在 MongoDB 存储库中存储正确的值。为此，可以使用一组绑定服务（通过 `vcapinate` 工具和 Wercker 服务镜像）启动服务器。服务器运行后，就可以向队列提交各种不同事件，等待处理器完成，然后查询存储库，以确保事件能安全生成。

如果想查看集成测试的代码，请访问 https://github.com/cloudnativego/drones-events/blob/master/integrationations /_test/integration_test.go。

构建查询处理程序服务

在完成了命令处理程序和事件处理器的创建后，本章中最难的部分就已经完成了。在本章中，查询处理程序是所有三个服务中最“蠢”的，它只查询 MongoDB 中最新的警报、遥测和位置事件。

当我们在事件处理器项目中创建 MongoDB 存储库（其代码在 GitHub 中，本章中没有展示）时，想要执行的相关查询就已经写好了。

这些代码都是比较简单的，之前已经写过很多次了。以下代码段是实现请求处理程序的，它能够返回给定无人机的最新遥测数据。

```
func lastTelemetryHandler(formatter *render.Render,
  repo eventRepository) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
        droneID := getDroneID(req)
        fmt.Printf("Looking up last telemetry event for drone %s\n", droneID)
        event, err := repo.GetTelemetryEvent(droneID)
        if err == nil {
              formatter.JSON(w, http.StatusOK, &event)
        } else {
              formatter.JSON(w, http.StatusInternalServerError, err.Error())
        }
  }
}
```

需要从每个处理程序的请求中提取出无人机的 ID，所以可以将该功能概括到以

下函数中。

```
func getDroneID(req *http.Request) (droneID string) {
        vars := mux.Vars(req)
        droneID = vars["droneId"]
        return
}
```

我们已经声明了 droneId 参数，并将其作为 REST 端点路由模式的一部分，服务器初始化函数中的代码如下所示。

```
func initRoutes(mx *mux.Router, formatter *render.Render, repo eventRepository) {
  mx.HandleFunc("/drones/{droneId}/lastTelemetry",
   lastTelemetryHandler(formatter, repo)).Methods("GET")
  mx.HandleFunc("/drones/{droneId}/lastAlert",
   lastAlertHandler(formatter, repo)).Methods("GET")
  mx.HandleFunc("/drones/{droneId}/lastPosition",
   lastPositionHandler(formatter, repo)).Methods("GET")
}
```

所有的查询服务本质上都是这样的。它被设计得尽可能简单，以便能够响应大规模的请求，并且可以支持高吞吐和低延迟。在更复杂或高级的实现中，查询服务通常由高速缓存服务来支持。

本章小结

这一章是本书中内容较多并且比较复杂的一章。值得高兴的是，我们已经构建了一支用 Go 语言实现的可以征服星球的无人机舰队，剩下的就只是下达命令了。

本章从事件溯源和 CQRS 模式的定义开始，探讨了最终一致性，讨论了这些模式与构建超大规模云原生应用程序的协作关系。

本章创建了一个命令接收器，用来接收新传入的命令，然后将它们转换为事件，并分派到队列中去。另外，本章还创建了一个事件处理器，从队列中读取事件并处理它们，执行必要的计算（虽然计算只是简单地捕获最近发生的事件），并把相应的事件存储到视图存储器中。最后，我们创建了一个查询服务，让消费者和应用程序能够查询事务的当前状态，而不是检查 reality 的快照。

撇清服务之间的关系并将它们适当地分离，这样有利于在单独扩展、更新和部署系统中的各个组件的同时保证其他组件还能正常运行，将宕机时间降低为 0。

9

使用 Go 构建 Web 应用程序

设计软件有两种方法：一种是使它足够简单以至于明显没有缺陷；另一种则是使它足够复杂以至于缺陷不那么明显。相比起来，第一种方法要困难得多。

——东尼·霍尔，1980 年图灵奖得主

Web 应用程序不过是一个服务，与其他服务不同的是，它与浏览器之间的隐性约定要求至少有一个服务资源以 HTML 的方式响应。

Web 应用程序经常带给人负面感受，五花八门的框架和库给工作增加了很多“黑魔法”和抽象风格。在有些技术圈子里，“Web 应用”常常会让人觉得是一个难以维护的庞然大物，但事实并不总是这样的。

在某些语言中，不能用构建一个服务时用到的库或 API 来构建网站，这会加深人们对于“网站是一个庞然大物”的错误印象。

本章将介绍以下几点。

- 用 Go 创建一个功能齐全的网站。
- 介绍如何处理静态文件和 asset。
- 结合已有的微服务知识为 JavaScript 客户端提供 RESTful 端点。
- 学习如何使用服务端模板。
- 学习如何处理 HTML 表单。
- 使用 Wercker 在云端构建并部署一个网站。

处理静态文件和 asset

100 万个 Web 应用程序有可能会进行 100 万种不同的操作，但是每个 Web 应用

程序都需要做的一件事就是提供文件服务。Web 应用程序需要向客户端的浏览器暴露一些文件，哪怕仅仅只有一个 `favicon.ico`

大多数现代 Web 应用程序使用层叠样式表（CSS）、JavaScript、图片甚至是特殊的字体（如 Awesome[1]字体）和字形库，这也是我们工具箱中最常用的库之一。

如果之前经历过在其他语言中从微服务转换到 Web 应用程序的过程，可能会认为必须要抛弃现有的代码并重新学习使用新的库或 API 的方法，幸运的是，在 Go 中不需要这样。

先创建一个目录结构，然后提供静态文件（包括 HTML 文件）。

- `static-content`
  - `js`
  - `images`
  - `css`

把 HTML 文件放到 `static-content` 目录下，把 JavaScript 文件放到 `js` 目录下。这里可以复用之前一直在使用的模板 `main.go` 和 `server.go` 来提供静态文件服务。实际上，可以直接复制 `main.go` 而不用做任何修改。

代码清单 9.1 展示了 `server.go` 文件的内容，其中将虚拟 URL "/" 映射到了 `static-content` 目录中。

代码清单 9.1 server.go

```
package main

import (
        "net/http"

        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
)

// NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {
  n := negroni.Classic()
  mx := mux.NewRouter()
```

1 可在 https://fortawesome.github.io/Font-Awesome/中查看关于 Font Awesome 的相关内容。

```
    initRoutes(mx)
    n.UseHandler(mx)
    return n
}

func initRoutes(mx *mux.Router, formatter *render.Render) {
    webRoot = os.Getenv("WEBROOT")
    if len(webRoot) == 0 {
            root, err := os.Getwd()
            if err != nil {
                    panic("Could not retrieve working directory")
            } else {
                    webRoot = root
            }
    }
    mx.PathPrefix("/").Handler(http.FileServer(http.Dir(webRoot + "/assets/")))
}
```

可以看到，我们基本上只添加了一行代码，用于在服务上配置 RESTful 路由。其中有几个嵌套的函数需要说明一下。

- `http.Dir`：`Dir` 在给定的路径上实现了一个 `FileSystem`，基于底层操作系统。本示例将在应用程序根目录下对外暴露 `assets` 目录。
- `http.Fileserver`：`FileServer` 返回一个 handler，该 handler 使用暴露的文件系统来响应 HTTP 请求。
- `mux.Router.PathPrefix`：添加一个路径前缀到路由匹配规则中。

继续在 `main.go` 和 `server.go` 所在的目录下创建一个 `assets` 目录，不要向该目录中添加任何内容，然后编译并运行 Web 应用程序。

首先来看一下在缺少文件的情况下会发生什么，并且确认使用默认的方法是否可以浏览目录。下一步，开始添加 HTML 文件和 CSS 样式表。

通过将前面服务示例中的 `initRoutes` 函数改写为仅使用一个函数调用的方式，就可以成功地创建一个基本的 Web 应用程序，是不是很简单！

支持 JavaScript 客户端

支持简单的文件服务是所有网站的基本功能，但是我们需要更多的功能来创建真正的 Web 应用程序。目前很难找到一个不使用 JavaScript 的网站，具体来说是不

太可能找到一个没有暴露 RESTful 端点给 JavaScript 使用的 Web 应用程序。

想要支持 JavaScript 客户端首先要能够支持 JavaScript 文件。前面创建的 `assets` 目录已经涵盖了该内容，下一步，需要创建一个使用这个脚本的 HTML 文件。

代码清单 9.2 是一个非常基础的 HTML 文件。该 HTML 中加载了一个名为 `hello.js` 的文件，其中包括 jQuery 和 JavaScript，并通过 Web 服务器提供本地服务。

在页面的主要部分，一共有两个段落。这些段落中包含了 CSS class。因此，可以通过编写 JavaScript 从服务器中获取数据来修改这些段落中的内容。

虽然这个例子是经过人为设计的，并且显得过于简单，但这个例子给我们传达了一种理念：他人试图说服我们只有使用其他语言才能完成的标准 Web 应用的构建，其实是如此简单。

代码清单 9.2 index.html

```
<html>
<head>
  <link rel="stylesheet" href="css/main.css" />
  <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.10.2/jquery.min.js">
  </script>
  <script src="js/hello.js"></script>
</head>
<body>
  Sample Go Application! <br/>
  <div>
    <p class="greeting-id">The ID is </p>
    <p class="greeting-content">The content is </p>
  </div>
  </body>
  </html>
```

我们还没有准备好运行任何程序，因为还没有编写 JavaScript 代码。如果大家熟悉 jQuery，对代码清单 9.3 中的代码应该就比较容易理解了。

代码清单 9.3 hello.js

```
$(document).ready(function() {
    $.ajax({
        url: "/api/test"
    }).then(function(data) {
        $('.greeting-id').append(data.id);
```

```
        $('.greeting-content').append(data.content);
    });
});
```

JavaScript 脚本在 HTML 文档加载完成后立即对 RESTful 接口（/api/test）执行一个 Ajax 调用，然后将返回的数据追加到代码清单 9.2 创建的段落元素中。这体现了 JavaScript 的巨大优势之一：不需要在某个实体中显式编码 JSON。JavaScript Object Notation （JSON）实际上起源于 JavaScript，因此 JavaScript 原生支持 RESTful 端点返回的数据。

经过在前面几章中多次创建 service 后，暴露一个端点对于我们来说应该已经是小菜一碟了。首先，像前面几章中那样创建一个 handlers.go 文件，如代码清单 9.4 所示。

代码清单 9.4 handlers.go

```
package main

import (
  "net/http"

  "github.com/unrolled/render"
)

type sampleContent struct {
  ID      string 'json:"id"'
  Content string 'json:"content"'
}

func testHandler(formatter *render.Render) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
        formatter.JSON(w, http.StatusOK,
        sampleContent{ID: "8675309", Content: "Hello from Go!"})
  }
}
```

可以修改 server.go 文件，将/api/test 端点映射到 initRouters 函数中的 handler 中。

```
mx.HandleFunc("/api/test",testHandler(formatter)).Methods("GET")
```

现在如果编译运行 Web 应用程序，将自动使用我们创建的 index.html 文件提供服务。然后通过运行 JavaScript，从服务端点获取数据，操作段落元素。

图 9.1 显示了该网页在 Chrome 浏览器中打开时的状态。

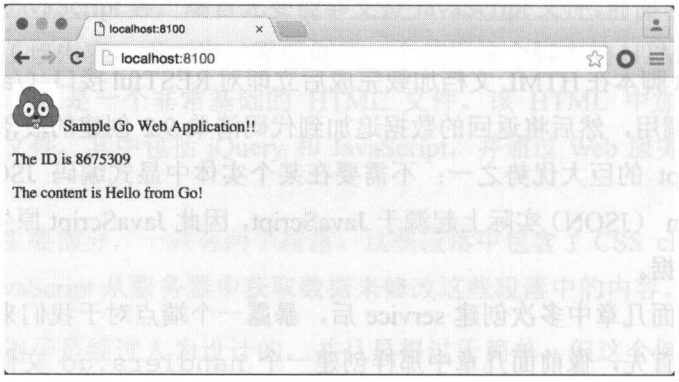

图 9.1 使用 JavaScript 和 REST 服务编写的 Web 应用程序

我们可以查看 Web 应用程序的控制台输出的 Negroni 追踪信息，获知 JavaScript 实际上是通过访问服务端点来生成图 9.1 所示的画面的。

使用服务端模板

在上一节中，我们讨论了如何构建和运行一个 HTML 页面，该页面包含 JavaScript，并且与 Go 应用程序暴露的端点交互。

我们认为这是大多数人构建现代 Web 应用程序的方式：暴露一个特别简单的 HTML，大多数工作都通过 AngularJS 等前端框架完成。

这可能是理想的情况，但总有一些时候，无论我们怎么努力，都难免将服务端数据渲染到 HTML 中来展现。

例如，有一个安全的网站，要创建一些包含当前登录用户用户名的 JavaScript 变量。在页面的 JavaScript 加载之前，还可能已经读取了一些 cookie 的值或其他想要放进 HTML 的数据。

实际上这是相当简单的。想要了解使用模板的完整示例，请查看 https://github.com/cloudnativego/web-application-template。

此示例的 `main.go` 文件与其他文件相同，我们要执行的所有操作都在代码清单 9.5 所示的 `server.go` 文件中。这个示例不是通过服务端点来暴露要动态呈现的数据，而是将该对象的引用注入到模板处理器中，以便在发送到客户端浏览器之前可

以操作 HTML。

代码清单 9.5 server.go

```
package main

import (
  "net/http"
  "text/template"

  "github.com/codegangsta/negroni"
  "github.com/gorilla/mux"
)

// NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {
  n := negroni.Classic()
  mx := mux.NewRouter()

  initRoutes(mx)
  n.UseHandler(mx)
  return n
}

func initRoutes(mx *mux.Router) {
  mx.PathPrefix("/images/").Handler(http.StripPrefix("/images/",
    http.FileServer(http.Dir("./assets/images/"))))
  mx.PathPrefix("/css/").Handler(http.StripPrefix("/css/",
    http.FileServer(http.Dir("./assets/css/"))))
  mx.HandleFunc("/", homeHandler)
}

type sampleContent struct {
  ID      string `json:"id"`
  Content string `json:"content"`
}

var t *template.Template

func init() {
  t = template.Must(template.ParseFiles("assets/templates/index.html"))
}
func homeHandler(w http.ResponseWriter, req *http.Request) {
  data := sampleContent{ID: "8675309", Content: "Hello from Go!"}
  t.Execute(w, data)
}
```

正如前面的例子所示，我们使用 PathPrefix 来提供虚拟的文件目录，如 images 和 css 目录。下面重点来看一下 homeHandler 函数。

在这个函数中，我们创建了一个 sampleContent 示例，然后调用 template 结构体的 Execute 方法（t 变量）。template 在包的 init 函数中进行初始化。

> **警告**
>
> 使用 init 函数需要注意。此函数应在包初始化之前被调用，并确保在应用程序的 main 函数之前被执行。但是，如果包中包含多个 init 函数，则不能保证它们的执行顺序。换句话说，永远不要在包初始化程序中依赖其他包初始化。

代码清单 9.6 展示了包含模板处理标记的 HTML，该 HTML 允许在发送到浏览器之前插入数据。除了替换语法，该段代码与前面的 index.html 几乎是一样的。

代码清单 9.6 index.html

```
<html>
<head>
  <link rel="stylesheet" href="/css/main.css" />
</head>
<body>
  <img src="/images/cng.png" height="48" width="48" />
  Sample Go Web Application!!
  <div>
    <p class="greeting-id">The ID is {{ .ID }}</p>
    <p class="greeting-content">The content is {{ .Content }}</p>
  </div>
</body>
</html>
```

现在运行这个应用程序，可以看到和上一个 JavaScript 示例几乎一模一样的输出结果，但是该程序中没有使用任何 JavaScript 和 RESTful 端点。

随着越来越多的 Web 应用程序使用 Angular、React 等框架，转向单页应用程序模型，对于服务器端的模板需求已经没有以前那么强烈了。

处理表单

回到创建强大 Web 应用程序的辉煌时代，开发的巅峰是表单处理。如果你知道如何解析和验证表单，那么你将会是 Web 开发团队中的重要成员。

Go 使得表单处理变得非常简单。表单值作为 request 结构体中的 Form 属性被暴露出来，以字符串到字符串切片的 map（map[string][]string）的方式被处理。

如果需要处理表单提交的数据，只需访问此集合并做出相应的处理即可。

示例中之所以没有包括手动处理表单的代码，是因为这样处理表单违反了后面将提到的一些模式规则。

如果使用像 React 这样的框架来构建极其强大、交互性强的页面，那么也将使用它来验证用户输入，然后将该输入提交到 Go 应用程序提供的 RESTful 端点上，并且不再允许用户使用老式表单的 POST 方法。

如果必须要手动处理表单，那么只需要访问请求结构体上的 Form map 即可，以下处理程序实现的功能是将表单键值对打印到控制台中。

```
func processFormHandler(w http.ResponseWriter, r *http.Request) {
    r.ParseForm()
    for k, v := range r.Form {
        fmt. Printf("Key %s, Val %s", k, strings.Join(v, ""))
    }
    fmt.Fprintf(w,"Form handled.\n")
}
```

注意

如果没有事先调用 ParseForm 或 ParseMultipartForm，则无法访问 Form 属性，具体取决于接受输入的表单类型。

使用 cookie 和会话状态

在过去流行的的开发方式中，似乎必须处理 cookie，它们现在依然被大多数会话状态管理系统使用。如果需要区分不同的浏览器，记住用户登录网站后所执行的操作，这些情况下可能会使用到 cookie，即使大家可能并不知道自己其实正在使用 cookie（见我们在本书中不断讨论的黑魔法）。

幸运的是，在 Go 中处理 cookie 很简单，cookie 已经成为标准库 net/http 中的一部分，所以不必再去寻找第三方包。

cookie 从请求中读取并被写入响应包。浏览器的工作是收集与向自己发出请求的网站相关的 cookie，并根据需要将 cookie 添加到每个请求中。当 Web 应用程序返回

的响应包含 cookie 时，浏览器必须写入该 cookie。

每个浏览器存储 cookie 的方式都不一样，而且同一个浏览器的存储方式在不同的操作系统下也会有所不同。虽然在用户磁盘上存储 cookie 可以编码其他网站和恶意用户访问，通常情况下是安全的，但仍然需要警惕。

一个好的方案是，不要在 cookie 中存储机密或敏感信息。不要在 cookie 中存储用户凭据等内容，如果必须存储唯一标识信息，请确保它即便是落入坏人之手也不会造成损失。还有一个好的方案是专门为 cookie 生成随机标识，并且该 ID 标识仅在服务器接收到验证请求后才会映射到有用的信息。

如上文所述，cookie 中的唯一标识通常用于识别会话状态之类的临时信息。

写入 cookie

写入 cookie 很容易。net/http 包中的 Cookie 结构体定义如下。

```
type Cookie struct {
        Name string
        Value string

        Path string          // optional
        Domain string        // optional
        Expires time.Time    // optional
        RawExpires string    // for reading cookies only

        // MaxAge=0 means    no   'Max-Age' attribute specified
        // MaxAge<0 means    delete cookie now, equivalently 'Max-Age: 0'
        // MaxAge>0 means    Max-Age attribute present and given in seconds
        MaxAge   int
        Secure   bool
        HttpOnly bool
        Raw string
        Unparsed []string    // Raw text of unparsed attribute-value pairs
}
```

cookie 的核心是一个名称/值对。可以进一步自定义和调整 cookie 来执行各种有趣的操作，这些操作在其他地方有很多文档，并且对应用程序云原生属性几乎没有影响。

编写一个处理程序，当使用 Web 浏览器访问 RESTful 端点时，可以获取到 cookie。这样的处理程序如下。

```
func cookieWriteHandler(formatter *render.Render) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
```

```
        expiration := time.Now().Add(2 * 24 * time.Hour)
        cookie := http.Cookie{Name: "sample",
         Value: "this is a cookie", Expires: expiration}
        http.SetCookie(w, &cookie)
        formatter.JSON(w, http.StatusOK, "cookie set")
    }
}
```

如上面代码所示，写入 cookie 仅需要一次函数调用，即调用 `setCookie`。

读取 cookie

读取 cookie 和写入一样简单。可以使用 `Cookies` 函数访问给定请求的所有 cookie 的切片，或者尝试访问单个 cookie，程序如下。

```
func cookieReadHandler(formatter *render.Render) http.HandlerFunc {
  return func(w http.ResponseWriter, req *http.Request) {
        cookie, err := req.Cookie("sample")
        if err == nil {
                fmt.Fprint(w, cookie.Value)
        } else {
                fmt.Fprintf(w, "failed to read cookie, %v", err)
        }
  }
}
```

在上述处理程序中，我们尝试检索一个名为 `sample` 的 cookie，并在浏览器中显示它的值。首先可以访问`/cookies/write` 端点，然后通过访问`/cookies/read` 来测试这个值，这些端点如下。

```
mx.HandleFunc("/cookies/write", cookieWriteHandler(formatter)).Methods("GET")
mx.HandleFunc("/cookies/read", cookieReadHandler(formatter)).Methods("GET")
```

使用 Wercker 构建和部署

到目前为止，我们一直在使用一个非常基础的 Web 应用程序，它对外暴露 JavaScript、HTML、CSS 和图像等静态文件。我们还了解了如何修改同一个服务，以便它可以暴露一个 API 给该应用程序页面中运行的 JavaScript 使用。

这很棒，但这只是一个开始。对于构建一个更大的应用程序来说，这只是冰山一角。在构建之前，应该确保应用程序在持续交付的流水线上。为此，与本书中所有的应用程序一样，我们将使用 Wercker。可以在 https://app.wercker.com/#applications/

56a4f473212b43b24e0badbb [译注1]和 GitHub 源码地址 https://github.com/cloudnativego/web-application 中查看此示例的 Wercker 构建定义。

我们使用 Glide 来管理这个程序的依赖，当使用 Wercker 来构建时，它提供了依赖版本锁定。在我们将思维模式从学习 Web 应用程序技术切换到构建真实功能之前，需要对资源进行测试。

以下是 `wercker.yml` 文件中构建 pipeline 的部分代码。

```
build:

  steps:
    - setup-go-workspace

    - script:
          name: go get
          code: |
            cd $WERCKER_SOURCE_DIR
            go version
            go get -u github.com/Masterminds/glide
            go get -u github.com/cloudnativego/cf-tools/vcapinate
            export PATH=$WERCKER_SOURCE_DIR/bin:$PATH
            glide install

    # Build the project - script:
    - script:
          name: go build
          code: |
            go build

    # Test the project
    - script:
          name: go test
          code: |
            go test -v $(glide novendor)

    - script:
          name: copy files to wercker output
          code: |
            cp -R ./ ${WERCKER_OUTPUT_DIR}
```

有了这段代码，我们便可以使用以下命令，将其作为一个自包含的 Docker 镜像

译注1 读者暂时无权限访问该示例。

来运行应用程序了。

```
$ docker run -p 8100:8100 cloudnativego/web-application:latest
[negroni] listening on :8100
```

如果是第一次下载镜像，我们将会看到一些控制台输出，显示正在下载到本地缓存的 Docker 镜像各个分层的信息。

本章小结

正确设计的 Web 应用程序只不过是暴露静态文件端点以及传统服务样式端点的微服务，本质上没有“云原生”或“非云原生”Web 应用程序的区别，但有些在云中构建 Web 应用的方法是非常离谱的。

本章中提供了一些简单的例子，大家可以结合学到的知识以及已具备的提供静态文件、模板处理文件、JavaScript 文件和图像等资源的能力，使用 Go 来构建服务。本章最后介绍了如何使用 Docker 和 Wercker 来自动构建和部署 Web 应用程序。

在本书后面的章节中，我们将讨论安全、Web sockets、模式和库（像 React 和 Flux）等技术点，那时这些 Web 基础示例就会被派上用场。

10
云安全

在没有发生安全问题之前，出于安全的考虑总被认为是多余的。

——Robbie Sinclair，澳大利亚新南威尔士州国家能源安全负责人

安全性实现起来是困难的。它混乱、复杂、费时间，而且相对于编写程序其他部分的代码，编写与安全相关的代码更加无趣。因此，大多数和安全相关的代码会被留到最后再编写，在上线的最后阶段通常会手忙脚乱，并且需要开发者做出权衡：是修复 bug 还是添加新功能？既然安全性还没有实现，那么就应该被当作新功能看待，就像好莱坞里常说的“*left on the cutting-room floor*”，本来是很好的镜头片段，却在影片发布前被抛弃在剪辑室的地板上。

事实上不一定是这样的，更重要的是，我们不能这样做。如果大家正在考虑构建云应用程序，扩展它们以支持海量数据，实现在世界各地运行，自动配置，无须人工干预即可自动启动，那么在构建这样的应用程序时，安全问题根本不能事后再考虑。

本章将讨论以下几点。

- 用于保护 Web 应用程序的选项和实现。
- 如何保护微服务。
- 关于隐私和数据安全的说明。
- 阅读练习，围绕开发安全服务建立更多的肌肉记忆。

保护 Web 应用程序

在这一节中，我们将学习如何保护 Web 应用程序。虽然并没有关于云原生安全

的本质上的说明，但是要保护部署在云上的 Web 应用程序，有一些做法和模式就一定要避免。

应用程序安全性选项

目前有很多种不同的方式可以保护应用程序。有十几种技术，每种技术通常有几个变体，有多个库和语言 API。以下列表远不完整，但是可以给大家提供一些选择和灵感。

- HTTP 基本验证：验证简单的用户名和密码，由客户端对其进行哈希散列操作并发送到 Web 服务器上。虽然 SSL 禁止窥探密码，但仍存在一些与基本验证相关的其他漏洞，包括浏览器缓存密码、每次请求中传输的密码等。
- 表单（cookie）：包含网页（而不是浏览器）产生的凭据。页面会对这些凭据进行验证并返回 cookie 作为响应。单独使用这种方法时，也存在安全问题。
- Windows 身份验证：从历史角度看，局域网和企业站点会优先选择这种方式，即使它们被部署在内部或私有云上。由于多种原因，这种方式不是实现云原生安全性的好的选择。
- SAML（ADFS、AD Azure、Shibboleth 等）：**安全断言标记语言**是一种 bearer token 技术，其中包含用于授权 SAML token 的身份信息提供者和验证它们的应用程序。SAML 是广泛用于联邦身份[1]的行业标准，并且拥有许多支持多种编程语言和操作系统的 API 及服务器产品。
- OAuth/OpenID：一种开放的授权标准，在允许用户使用第三方凭据（如 Facebook、Google、Twitter 等）登录的网站中非常常见。使用 OAuth 的效果非常好，本章中我们将使用它实现云原生安全。
- 习惯性：那些不愿意从昔日的错误中吸取教训的人将来注定会重蹈覆辙。

> **警告**
>
> 不要依赖底层操作系统中的任何功能、密钥库或用户账户信息的安全机制。如果想要将它们作为支持服务，那么可以依赖它们。系统安全性需要实现，而无须考虑运行它的虚拟主机是否能正常运行。

1 参考 https://en.wikipedia.org/wiki/Federated_identity 中的内容。

在本章的后续部分，我们将开发两个代码示例：一个安全的 Web 应用程序和一个安全的微服务。

首先，使用 OAuth 作为网络应用授权机制，选择它有以下几个原因。

- 无须关注底层操作系统或已经安装的功能。
- 为了允许在 Web 上的不同系统中共享授权声明而特别设计。
- 简单易用。
- 坚实而可靠。
- 提供简单、易于使用的 Go 语言库。
- 提供免费、开源的身份信息提供商服务器软件。
- 有权使用基于云的身份信息提供商，如 Google、Auth0 和 StormPath。

设置 Auth0 账户

如果没有服务授予 JWT（JSON Web Token，一种用于表达可在多方传输的 URL 安全 JSON 格式），OAuth 则无法工作。身份信息提供商负责验证用户的身份并返回一个合法的 JWT 响应。然后，应用程序可以轻松拆解 JWT 格式，获取访问用户配置信息的权限，而不需要用户再次输入密码。

可以使用 Google Web Applications 设置一个身份信息提供者，还可以使用来自 Cloud Foundry 中的开源 UAA 服务或 Pivotal Cloud Foundry Single-Sign-on。本书的目的是让操作尽可能简单，让大家尽可能少地使用信用卡，因此我们将使用一个名为 **Auth0** 的服务。Auth0 是一个云安全提供商。

可以配置一个 Auth0 账户来维护一个私有用户数据库（允许用户自己注册），或者通过配置使我们的网站接受第三方身份验证，例如使其允许通过 Twitter、LinkedIn、Facebook、Google 和许多其他应用程序账户进行登录。在本书中，我们选择使用私有数据库。

要创建 Auth0 账户，请访问 http://auth0.com 并阅读网站提供的服务。如这本书中所有其他服务一样，可以直接注册而不需要提供信用卡。

我们可以选择任何自己喜欢的身份验证来源，本章其他部分的代码示例不需要任何更改即可运行，这也是我们喜欢将外部身份验证用于云应用程序的原因之一。登录后，可以看到如图 10.1 所示的仪表板界面，Auth0 为身份验证系统提供了大量的度量指标、仪表板和控制认证系统的功能。

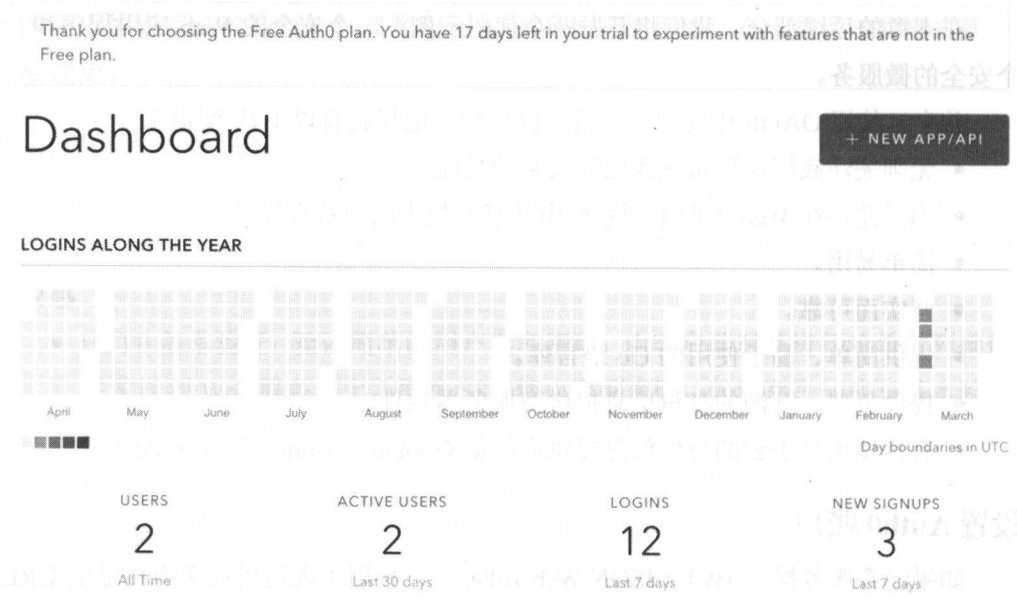

图 10.1 Auth0 仪表板

来源：http://auth0.com

构建一个 OAuth 安全的 Web 应用程序

现在已经设置了 Auth0 账户，我们还需要创建一个基于它的应用程序。当遇到未进行验证的请求时，该程序可以将请求重定向到 Auth0 中以获取有效的令牌。

如果收到一个包含令牌的请求，则需要打开该令牌，并将该用户的信息暴露给处理器程序。以下例子中有很多步骤，我们的目标是让它足够简洁，这样就可以做到专注于重点。

可以在 https://github.com/cloudnativego/secureweb 上找到以下程序的完整源码。我们需要做的第一件事是建立一个 server，代码清单 10.1 中的程序在本书中已经实现过很多次了。

代码清单 10.1 server/server.go

```
package server

import (
        "net/http"
```

```
        "github.com/astaxie/beego/session"
        "github.com/cloudfoundry-community/go-cfenv"
        "github.com/cloudnativego/cf-tools"
        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
)
type authConfig struct {
        ClientID        string
        ClientSecret    string
        Domain          string
        CallbackURL     string
}

func NewServer(appEnv *cfenv.App) *negroni.Negroni {
  authClientID,_ := cftools.GetVCAPServiceProperty("authzero", "id", appEnv)
  authSecret,_:= cftools.GetVCAPServiceProperty("authzero", "secret", appEnv)
  authDomain,_:= cftools.GetVCAPServiceProperty("authzero", "domain", appEnv)
  authCallback,_:= cftools.GetVCAPServiceProperty("authzero", "callback",appEnv)

  config := &authConfig{
        ClientID:       authClientID,
        ClientSecret:   authSecret,
        Domain:         authDomain,
        CallbackURL:    authCallback,
  }

  sessionManager, _ := session.NewManager("memory",
   '{"cookieName":"gosessionid","gclifetime":3600}')
  go sessionManager.GC()

  n := negroni.Classic()
  mx := mux.NewRouter()

  initRoutes(mx, sessionManager, config)

  n.UseHandler(mx)
  return n
}

func initRoutes(mx *mux.Router, sessionManager *session.Manager,
  config *authConfig) {
  mx.HandleFunc("/", homeHandler(config))
  mx.HandleFunc("/callback", callbackHandler(sessionManager, config))
  mx.Handle("/user", negroni.New(
        negroni.HandlerFunc(isAuthenticated(sessionManager)),
        negroni.Wrap(http.HandlerFunc(userHandler(sessionManager))),
  ))
```

```
  mx.PathPrefix("/public/").Handler(http.StripPrefix("/public/",
    http.FileServer(http.Dir("public/"))))
}
```

在上面的例子中，大家可能注意到一个和之前的示例不同的地方，即用到了会话状态管理。这里使用第三方库启用 cookie-keyed 会话状态系统。我们将使用这个会话状态来记住从授权令牌中获取到的信息，这样就可以在渲染网页的时候使用这些信息了。

书中列出的源代码是直接从 Auth0 示例中复制的，故不包括回调处理程序。大家可以在 GitHub 仓库中找到该代码。

会话状态警告

为了使这个例子重点关注在安全性上，我们使用了最简单的会话状态管理方法：内存缓存。生产环境的应用程序需要能扩展到多个示例中，故这类会话状态管理方法并不适用。生产应用程序需要使用外部缓存（例如 Redis）、cookie 或某些其他组合，以进行适当地扩展。

接下来要做的是使用中间件。Negroni 包含中间件的概念，允许我们在处理函数执行前后注入其他处理程序，并使用代码（如验证检查）来封装处理程序。在代码清单 10.1 中，我们使用了 `isAuthenticated` 中间件来封装 `userHanlder` 函数，这将在 `userHanlder` 函数被调用前强制进行身份验证。

在其他框架的语言（如 Java 和.NET）中，大家可能已经见过支持类似功能的属性或注释。

代码清单 10.2 展示了中间件函数 `isAuthenticated` 的代码。

代码清单 10.2 server/middleware.go

```
package server

import (
        "net/http"

        "github.com/astaxie/beego/session"
        "github.com/codegangsta/negroni"
)

func isAuthenticated(sessionManager *session.Manager) negroni.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
```

```
        session, _ := sessionManager.SessionStart(w, r)
        defer session.SessionRelease(w)
        if session.Get("profile") == nil {
                http.Redirect(w, r, "/", http.StatusMovedPermanently)
        } else {
                next(w, r)
        }
  }
}
```

这段代码非常简单。每个包装了该中间件的处理器都会从会话中提取通过验证的用户身份信息的 profile。如果不存在，则重定向到主页。

如代码清单 10.3 所示，主页处理程序中包含一些 JavaScript，通过服务器端模板来创建一个登录按钮。这个按钮使用户转到 Auth0 中进行登录或注册，当用户从 Auth0 中重定向回来时，将会有一个身份验证令牌。

代码清单 10.3 server/home_handler.go

```
package server

import (
  "net/http"
  "text/template"
)
var bodyTemplate = '
   <script src="https://cdn.auth0.com/js/lock-8.2.min.js"></script>
   <script type="text/javascript">
   var lock = new Auth0Lock('{{.ClientID}}', '{{.Domain}}');
   function signin() {
     lock.show({
     callbackURL: '{{.CallbackURL}}'
     , responseType: 'code'
     , authParams: {
     scope: 'openid profile'
   }
   });
   }
    </script>
    <button onclick="window.signin();">Login</button>
'

func homeHandler(config *authConfig) http.HandlerFunc {
   return func(w http.ResponseWriter, r *http.Request) {
      t := template.Must(template.New("htmlz").Parse(bodyTemplate))
      t.Execute(w, config)
```

```
    }
}
```

从 Auth0 中返回的认证令牌将由 `callbackHandler` 函数解析。这个函数会打开令牌，验证它，然后将会话状态中的用户信息保存到用户 profile 对象中。如我们前面提到的，这是 Auth0 中的示例。

代码清单 10.4 server/user_handler.go

```
package server

import (
        "fmt"
        "net/http"

        "github.com/astaxie/beego/session"
)

func userHandler(sessionManager *session.Manager) http.HandlerFunc {
  return func(w http.ResponseWriter, r *http.Request) {
        session, _ := sessionManager.SessionStart(w, r)
        defer session.SessionRelease(w)

        // Getting the profile from the session
        profile := session.Get("profile")
        fmt.Fprintf(w, "USER DATA: %+v", profile)
  }
}
```

在代码清单 10.4 中，我们利用了存在于会话中的用户配置文件。这个非常简单的示例只是输出了用户 profile 数据，对于成熟的 Web 应用程序来说，用户 profile 数据将作为一个对象暴露给服务器端模板处理程序。

上面的寥寥几行代码就可以说明，使用 Go 和 OAuth 也可以像使用其他语言或框架一样轻松地保护 Web 应用程序。如果有朋友告诉你，Go Web 应用程序无法满足配置简单和强大的安全性的需求，你应该远离他们，生活中不需要这样的朋友。

运行安全的 Web 应用程序

运行应用程序最简单的方法是从 GitHub 中克隆，地址是 https://github.com/cloudnativego/secureweb，在终端中输入以下内容。

```
./runlocal
```

这是我们编写的一个脚本，允许在 Wercker Docker 镜像中运行程序，通过使用一些模板可以为 Auth0 账户注入值。

如果没有指定任何运行参数，它会显示一些帮助信息，说明如何创建一个应用程序运行时所需要的环境文件。为了实现这个功能，需要在 `local_config` 目录中创建一个名为 `env` 的文件，具体如下。

```
X_AUTHZERO_ID=(your auth0 ID)
X_AUTHZERO_SECRET=(your auth0 secret)
X_AUTHZERO_DOMAIN=(your auth0 domain, e.g. foo.auth0.com)
X_AUTHZERO_CALLBACK=http://192.168.99.100/callback
```

Docker machine 默认的 IP 地址是 192.168.99.100，即我们设置的回调地址（记住 Wercker 构建在 Docker 镜像中运行）。

在此之前需要设置一个 Auth0 账户。最简单的方法就是允许它使用社交网络身份验证，这样就不必配置自己的数据库了，但是必须确保将创建的数据库链接到 Auth0 中创建的应用程序上（官网上有详细文档）。

最后，需要将 `X_AUTHZERO_CALLBACK` 的值添加到应用程序授权的回调 URL 列表中。

现在运行刚刚创建的应用程序。

```
$ ./runlocal local_config/env
```

当屏幕上出现很多 Wercker 和 Docker 的输出信息后，应用程序将监听 80 端口。打开浏览器访问 http://192.168.99.100（Docker machine 的 IP 地址），应该可以看到一个简单的登录按钮。点击这个按钮，得到如图 10.2 所示的页面。

在完成注册或使用已有的账户登录后，进程将通过回调地址重定向回我们的应用程序。当回调处理程序完成 OAuth 令牌处理并填充会话状态后，会重定向到 URL `/users` 上。

用户处理程序会转储会话状态到用户 profile 中，所以我们可以看到对应用程序开放的各种信息。特别使人感兴趣的是 `alias`、`email` 和 `picture` 字段，这都是安全网站上非常宝贵的用户个性化体验。

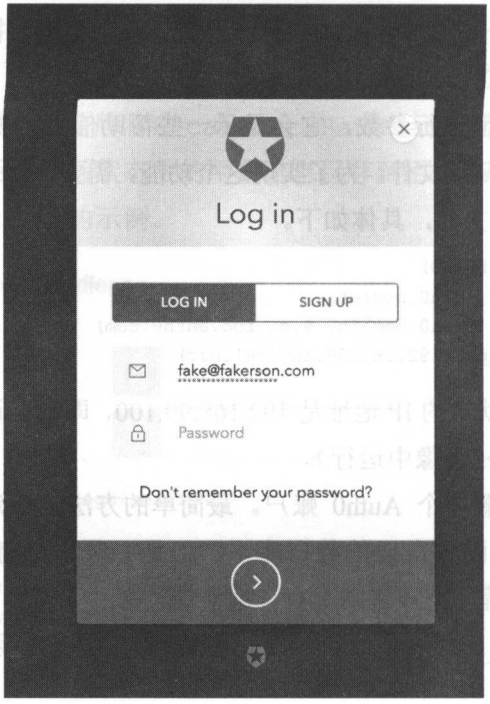

图 10.2 使用 Auth0 登录应用

来源：Auth0

保护微服务

在本节中，我们将介绍如何保护微服务。我们在之前介绍了如何轻松地搭建一个包含 OAuth 身份验证的 Web 应用程序，但这依赖于 Web 应用程序和身份验证提供商之间的重定向。

在下一节中，我们将讨论如何保护没有任何 UI 界面的微服务，并且不采用任何需要重定向的身份验证方案。

客户端凭据模式概述

服务的认证模式就像世界上的架构师一样多。开发人员经常会自己实现仅适用于他们公司的令牌处理和验证系统。

很多人使用客户端证书、加密或令牌、证书和加密三者的某种组合，在这里我

们将使用最简单的模式：客户端凭据。

注册成为 Auth0 服务的用户后，便获得了一个 **API key** 和 **API secret**。通过这些便可以使我们的服务与 Auth0 服务进行通信。当使用 Google Maps API、Facebook API、Twitter 的 API 或其他 API 时，我们运行在一个非常简单的模式下，代码必须使用 key 和 secret 配对来进行身份验证。

这样做的结果是，需要验证服务的人类消费者与消费服务的客户端代码之间将存在差异。这里有两个问题，可能会存在一些分歧。

- 最初发起请求的人的身份是什么？
- 是服务请求授权这样做的吗？

在某些情况下，服务不需要关心使用者的身份。例如，返回单个 SKU 仓库可用性服务时，不需要关心请求是由个人还是系统其他部分发出的。事实上，最佳的做法是，这种服务根本无须关心它是在为个人还是其他服务工作。当然，凡事总有例外。如果服务使用了 RBAC（基于角色的访问控制），那么就需要了解请求发起者的身份，这样才能确定是执行该请求还是通过返回 ***401 Unauthorized*** 或 ***403Forbbiden*** **状态码**来拒绝请求。

> **提示：关于 403 和 401**
>
> 当凭据不足时，经常可以看到安全的 RESTful 服务返回 401，这不是 401 代码的正确使用方式。当试图访问被保护的资源但未提供凭据时，才应该使用 ***401Unauthorized*** 状态码。当试图访问被保护的资源并且提供了正确的凭证，但是调用客户端没有足够的权限来执行请求时，应该使用 ***403 Forbidden*** 状态码。这是一个微妙但却很重要的区别，在保护 Web 服务时要记住这一点，这可能会为我们省去许多麻烦，尤其是在构建公共 API 时。

在许多企业生态系统中，可能需要在整个服务调用链中携带调用者的身份信息。出于合规性和审计目的，有时候这是必要的，但是应该尽量避免服务间的紧密耦合，除非实在没有其他办法。

在客户端凭据模式中，客户端每发出一次请求都会将一对值传递给服务，这对值通常被命名为 *key* 和 *secret*，也可以被命名为 *username* 和 *password*。出于和前面同样的考虑，我们一般避免使用后者：不应该假设有什么人出于什么样的意图而调用服务，我们要做的只是验证和响应请求。

使用客户端凭据保护微服务

我们已经了解了如何实现一个中间件，使其解析由 OAuth 身份提供商颁发的 JSON Web Token（JWT），以验证 Web 服务请求。

同样，可以使用中间件模式来保护微服务路由，唯一的区别是中间件的具体实现。下面这个简单的客户端代码展示了如何设置使用客户端凭据进行验证的安全根路径（`/api/*`）。

代码清单 10.5 server/server.go

```
package server

import (
        "github.com/codegangsta/negroni"
        "github.com/gorilla/mux"
        "github.com/unrolled/render"
)

//NewServer configures and returns a Server.
func NewServer() *negroni.Negroni {
  formatter := render.New(render.Options{
        IndentJSON: true,
  })

  n := negroni.Classic()

  // Public Routes
  router := mux.NewRouter()
  router.HandleFunc("/", homeHandler)

  // Protected API Routes
  apiRouter := mux.NewRouter()
  apiRouter.HandleFunc("/api/get", apiGetHandler(formatter)).Methods("GET")
  apiRouter.HandleFunc("/api/post", apiPostHandler(formatter)).Methods("POST")

  router.PathPrefix("/api").Handler(negroni.New(
        negroni.HandlerFunc(isAuthorized(formatter)),
        negroni.Wrap(apiRouter),
  ))

  n.UseHandler(router)
  return n
}
```

在代码清单 10.5 中，我们设置了安全性，保证了与 `apiRouter` 绑定的所有路

由都将由中间件函数 isAuthorized 来保护。这个模式看起来很熟悉，它在 Go 中非常常用。

接下来，代码清单 10.6 展示了 isAuthorized 函数的实现方法。

代码清单 10.6 server/middleware.go

```
package server

import (
        "net/http"
        "os"

        "github.com/codegangsta/negroni"
        "github.com/unrolled/render"
)

func isAuthorized(formatter *render.Render) negroni.HandlerFunc {
     apikey := os.Getenv(APIKey)
     return func(w http.ResponseWriter, r *http.Request, next http.HandlerFunc) {
           providedKey := r.Header.Get(APIKey)
           if providedKey == "" {
             formatter.JSON(w, http.StatusUnauthorized,
               struct{ Error string }{"Unauthorized."})
           } else if providedKey != apikey {
             formatter.JSON(w, http.StatusForbidden,
               struct{ Error string }{"Insufficient credentials."})
           } else {
               next(w, r)
           }
   }
}
```

此函数使用环境变量来确定认证所需的 API key 的预期值。在实际应用中，我们将同时比较认证客户端的 *key* 和 *secret*。

关于 SSL 的注意事项

敏锐的读者可能已经发现，我们没有编写任何一行和 SSL 相关的代码，关于这一点，有一个很好的理由：我们讨厌编写 SSL 代码。更重要的是，大家也会认同这一点。

有很多处理 SSL 证书验证和 SSL 终端的基础架构工具，它们已经无处不在，将微服务中的代码紧密地耦合到特定的 SSL 实现或规则上，基本上成为了反模式。

无论是利用SSL终端在反向代理后面手动部署代码，还是将代码部署到包含SSL终端的Cloud Foundry中，应用程序都不应该访问底层SSL的详细信息，除非可以明确地列出客户端证书的白名单，即使这样，这些工作也可以在微服务外的网关中来完成。

隐私和数据安全

安全性这个术语已经不堪重负。有无数类型的安全性，我们已经讨论了其中主要的类型——应用程序和服务的安全性。应该确保只有经过身份验证的用户可以访问Web应用程序，以及只有使用客户端凭证的用户可以访问服务。

最近经常在新闻中出现的另一种类型的安全是**数据安全**。任何不涉及数据安全的关于在云中构建大规模安全应用程序的讨论都是不完整的。

曾几何时，企业总是把数据中心设立在偏远隐蔽的地方，由训练有素的人员日夜把守，希望以此规避安全隐患。另外，企业还在网络的每个组件上增加防火墙，锁定数据库、打印机以及它们之间的一切通信。这就好比好莱坞让我们相信，秘密在杰森·布恩[译注1]和詹姆斯·邦德[译注2]的世界中可以得到保护。

这种“躲在墙后”的心态自有其优点，但它真的适用于云时代吗？如果没办法在物理上接触到自己的服务器该怎么办？当一半的基础设施可以无缝地从纽约迁移到内布拉斯加州再到加利福尼亚州，或者当数据库运行副本的数量可以动态扩展时，我们还会请一个穿制服的保安在门外看守吗？

黑客不能得到你没有的

黑客入侵的一个通行原则是，只要有足够的时间和资源，任何基础设施都可以被攻破。如果为了防止黑客入侵而做好最坏的打算，却不去思考究竟是什么导致了黑客入侵，那么你们公司的名字将赫然出现在各大报纸和社交媒体的版面上，而丢失信息的客户也将流失。

如果有人渗透你的服务，并以某种方式获得未经授权的访问，将会发生什么？

译注1　电影《谍影重重》中的男主角。

译注2　“007系列电影”中的男主角。

那时又该做些什么？如果一直遵循云之道，并且开发应用程序时坚持遵循云原生应用程序的 12 要素法则，那么我们将构建一个使用外部化配置的无状态服务。换句话说，对服务的内存空间不会有任何影响，入侵者无法获取用于和后端服务进行通信的 URL 和凭据。

假设有人进入我们的数据库，并且有查询所有数据的完全访问权限，那么他们会发现什么？对他们来说是否有利可图？

阻止数据库被窃取的关键不是让这种行为变得毫无可能，而是应该使数据库本身变得无利可图。如果黑客需要花太多时间去寻找想要的东西，他们就会转移目标。当他们进入数据库，发现所有有用的信息都被加密（并且密钥被存储在别的地方），或被存在另外一个数据库中，那么他们可能会转移目标。

以下是设计云上安全系统的一些基本原则。这个列表不是最全面的，如果对信息安全感兴趣，可以进行更加深入的学习，本书中不会进行过于深入的讨论。

- 不要在可能被泄露信息的人的数据库中存储重要信息。例如，如果必须存储信用卡号码，请不要将其存储在它的所有者的数据库中。更好的做法是，当客户提供信用卡时，使用第三方交换卡号，以获得在脱离该应用程序上下文的情况下没有任何意义或价值的令牌。
- 如果黑客有利可图或者可能破坏字段的内容，那么请在存储和传输过程中都进行加密。有一些现成的软件可以让数据在整个传输链路中始终保持加密状态，直到其到达拥有者的手中。
- 永远，永远，永远不要在应用程序的加密数据旁保存解密密钥。这就像把房门钥匙放在花盆里，然后张贴一个标志告诉大家：“不要往花盆里看，这里面有秘密。”
- 不要向 STDOUT 或 STDERR 发出 PII（个人身份信息）。在得知有许多攻击都是通过嗅探相对不安全的日志流而不是直接破解实际系统所产生的之后，大家一定会对此感到震惊。开发人员喜欢发出详细的诊断信息（如在库中存储记录等），并使用快捷方式（如 Go 中的%+v 标记）将记录的全部内容（包括字段名称）转存到日志流中。
- 为了实现无状态性和安全性，不要在内存中长时间存储机密信息，应该在处理完成后立即销毁。

另外，还有一个 3R 法则，虽然这不是一条固有原则，但却是我们工作时信奉的哲理，具体如下。

- **轮换**（Rotate）、**翻修**（Repave）、**修复**（Repair）：Justin Smith 的文章中讨论了每隔几分钟便轮换数据中心凭据、经常修复服务器以及在补丁程序可用后立即修复操作系统这几个概念。

我曾经与一位出色的安全专家聊过这个话题。他表示，想要从他的系统中获得任何有意义的数据，黑客必须侵入三个不同区域的三个数据库，以及在第四个位置获得加密密钥。如果有人能够侵入这么多系统，那么几乎不可能只是因为信息安全的问题，一定还有其他原因。

读者练习

本章构建了一个由 OAuth 使用第三方身份提供者 Auth0 来保护的 Web 应用程序，还构建了一个安全的 Web 服务，通过查询 API 密钥来验证客户端。

在读者练习中，希望大家能够完成这个示例，将安全 Web 应用程序与安全 Web 服务整合，尽可能模拟常见的实际场景。

1．将资源添加到安全服务中，并且提供一条有关使用该资源的用户的信息。例如，可以创建像`/customer/email address/orders`这样的资源集合，输出由该电子邮件地址指代的客户的订单信息。请确保服务的 JSON 响应中包含了电子邮件地址，以便知道它可以适用于不同的客户。

2．向安全 Web 应用程序中添加一个网页，显示从安全 Web 服务获取到的信息。例如，可以创建一个“我的订单”页面，用于显示从安全服务中检索到的订单列表。

（1）为了实现这一点，需要从会话状态中提取用户的电子邮件地址（现有示例中已经有一个访问此数据的例子），并使用它来构建对该安全服务的 REST 请求。

（2）还要创建一个 user-provided service，以便后台服务的 URL 不会在 Web 应用程序中进行硬编码。本书中无论是否用到 Cloud Foundry，都使用了这种获取依赖服务 URL 的模式。

完成练习后，大家一定会对构建依赖于后台服务的 Web 应用程序有一个非常好的理解。另外，还会对确保 Web 应用程序和服务的安全性所需的工作及其复杂性有很好的理解。

本章小结

安全性永远不应该是最后才考虑的事情。即便应用程序在没有安全认证的情况下也可以使用，但这一点必须是明确的——任何关于应用程序和数据安全性的决定都不能含糊。

本章介绍了关于如何在云中保护应用程序的概念，并讨论了一些比较常见和主流的策略，如 bearer token。

虽然本章没有涵盖与安全相关的所有主题，但在读完本章后，希望大家能对安全性有一个基本认识，并在此基础上引申出更多关于保护云中应用程序和服务安全性的更深层次的问题。

11

使用 WebSockets

简单是可靠性的先决条件。

——Edsger W. Dijkstra

上了年纪的人应该会记得，曾几何时，若希望在 Web 上看到最新的数据，必须点击浏览器上的**刷新**按钮。直到最近，网站才开始真正采用交互的方式，根据用户的需要自动更新网页。

尽管这是近来才有的创新，但它目前已无处不在。几乎每个使用 Web 应用程序的人都要用到这样的特性。如果 Web 应用程序无法动态更新信息，通知用户相关数据的重大变更，那么客户就会流失。如果修改了 Web 中的数据，而浏览器却并不知情，那么我们可能会改用其他的 Web 应用程序，然后在 Twitter 上发泄不满。

WebSockets 和服务器发送事件（Server-Sent Event，SSE）这两种技术能够提供更直观的交互式浏览器体验。在本章中，我们将讨论这两种技术，重点了解它们对云原生开发会产生怎样的影响。

本章的主要内容有以下两方面。

- 剖析 WebSockets 及其在云中的适用性。
- 使用第三方消息程序构建 WebSockets 式的应用程序。

WebSockets 解析

WebSockets 给人的感觉可能就像是奇妙的 UI 端一样，对于将大部分时间花在服务器和后端上的开发人员来说，可以放心地忽略它。但不幸的是，事实并非如此。目前的 Web 应用程序，甚至是最简单的应用，都在使用 WebSockets。

正当你查看 Facebook 的消息流，感叹着他人的生活是多么的优越时，紧跟着收到一条通知，显示有人点赞了你昨天发布的有趣的猫咪视频，这其实都得益于 WebSockets（或 SSE）。

当我们从 Domino 的网站上订购披萨时，从预定阶段到披萨完成再到最后递送到我们手中，这期间都无须刷新浏览器就可以看到进度条的变化，这就是 WebSockets 的功劳（同时也可能是社会衰落的象征）。

WebSockets 如何工作

由于 Web 页面需要具有在服务器和浏览器之间进行双向通信的能力，因此 WebSockets 仍然需要在不破坏传统的“request only”Web 模型的情况下，通过某种方式运行在现有的 HTTP 标准之上。

WebSockets 通过更新 HTTP 连接的方式运行。它总是由客户端通过命中服务器端点的的方式发起，`Connection: Upgrade` 头部信息的代码如下所示。

```
GET /game HTTP/1.1
Host: server.mygame.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: x3JJHMbDL1EzLkh9GBhXDw==
Sec-WebSocket-Protocol: awesomegame
Sec-WebSocket-Version: 13
Origin: http://mygame.com
```

这里面有一些 WebSockets 专有的的头部信息，例如密钥、协议和版本。密钥和版本由顶层的 WebSockets 协议使用。`Sec-WebSocket-Protocol` 头部信息表示允许客户端与服务器建立自定义握手，以便协定新 WebSockets 通道的 *content* 标准或定义。在这个示例中，我们使用了 `awsomegame` 协议。如果服务器能够解析该协议，那么它将做出相应的响应。虽然这无法解决服务器和客户端之间的误匹配问题，但是给了我们在做实际工作前尝试和设置层级的机会。

接着我们会从服务器中得到一个响应，指示连接已经从常规 HTTP 连接升级成为 WebSockets 连接。

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: HSmrc0sMlYUkAGmm5OPpG2HaGWk=
Sec-WebSocket-Protocol: awesomegame
```

现在我们已经打开了一个 HTTP 连接，并使用它来回传递 WebSockets 流量，而不是传统的 HTML 内容。当然，我们并不会手动编写 JavaScript 来建立这种连接更新，而是直接使用标准的 JavaScript WebSockets 库（或者其他任何我们喜欢的提供该包装器的 JS 框架）来编写代码。

```
var socket = new WebSocket('ws://server.mygame.com');
```

有了套接字变量后，我们就可以定义函数了，该函数可以在每次将数据从 server 端推送到浏览器中运行的 JavaScript 时被调用，接着可以使用 `send` 方法将数据发送到服务器端。

WebSockets 与服务器发送事件对比

服务器发送事件（SSE）相较于 WebSockets 有些许不同，它更轻量，但却是单向的。顾名思义，服务器发送事件只能用于从服务器端向浏览器端推送数据。因此，许多大型 Web 应用程序通常倾向于采用这种方式，而不是使用可以全双工通信的 WebSockets。

WebSockets 和 SSE 之间的一些关键区别如下。

- WebSockets 是全双工的；SSE，顾名思义，只能由服务器端发送。
- 由于 connection upgrade，WebSockets 的流量不包含在传统的 HTTP 连接中，这在遇到旧版本的路由器和代理时可能会出现问题 。
- 应用程序越复杂，SSE 需要的 HTTP 连接就越多，而对于很多 App，不论它们有多复杂，只要为每个用户维持一个连接即可。

在本章中，我们将讨论 WebSockets，但绝大多数的讨论和设计模式同样适用于 SSE。

设计 WebSockets 服务器

假设确定采用 WebSockets，并且想要给应用程序前端的 JavaScript 提供 WebSockets 功能。如果使用现代的服务器端 Web 开发框架来实现，那么将会有一个“简单按钮”，使用它可以在编写处理函数（或控制器方法，如果使用 MVC 模式的话）时少写很多的代码。即使有这样一个简单按钮，还是要保证以下几点。

- 支持连接升级。
- 支持处理全双工 WebSockets 网络连接。

- 保持所有 WebSockets 连接的某种映射。
  - 循环该映射，发送“将要广播一条消息”的数据给网站上的所有用户。
  - 查找特定的套接字（根据用户匹配），能够将消息传递给单个目标。例如，通知你又有人赞了你发布的猫咪视频。
  - WebSockets 连接关闭时清理映射。

WebSockets 的云原生适应性

我们已经了解了 WebSockets 以及为了支持它需要在服务器端做的工作，那么 WebSockets 是如何在云端运行的呢？

首先，让我们来看一下**无状态**。所有真正的云原生应用程序都是无状态的。事实上，为了简化 WebSockets 服务器的功能，不得不保持对所有打开的 WebSockets 的引用，这样做会有一定风险。当我们开始给这些能够辨识个人用户的套接字添加映射时，情况将变得更糟。

如果应用程序关闭，所有的 WebSockets 连接都会丢失。大家可能都遇到过这种情况，却不知道它发生的深层原因。假如我们正在刷新社交媒体的消息流，但它却突然像停止更新一样：没有任何新消息！猫咪停止弹钢琴了！为了在全面崩溃前进行最后的挽救，我们尝试点击刷新按钮，幸运的是，页面恢复正常。以上是 WebSockets 连接丢失的典型案例。

由于 WebSockets 的有状态性，当对其进行水平扩展时，会产生一些隐患。为了应对日益增加的负载，能够对应用程序的实例数进行动态扩展，因此我们选择在云中部署应用程序。不幸的是，这会导致 WebSockets 出现问题。

假设我们正在和朋友一起玩一款多玩家浏览器端游戏，我们可以轻而易举地取得胜利。和朋友之间可以进行无障碍沟通，包括进行游戏的时候可以发送具有侮辱性的私信，这将是一种很棒的体验。

假设该游戏变得非常流行，我们在云端同时运行该应用程序中的 8 个示例，这时会发生什么呢？该游戏是一个单页面的应用程序，它大部分的通信都基于 WebSockets，还有一些通过 REST 调用响应 WebSockets 的消息来增加数据。

假设我们在**示例一**上登录并重新开始游戏，我们的朋友在**示例四**上登录。当他向我们发送私信时，示例将通过检查内存映射来查找属于我们的套接字。因为套接

字在**示例一**中，所以它无法被找到。像这样的错误可能会被隐藏，或在朋友的游戏中通过怪异的方式表现出来。最终的结果是，尽管双方都连接到游戏，并且双方客户端都认为自己与服务器建立了正常的连接，但双方玩家之间却无法通信。

此架构如图 11.1 所示。

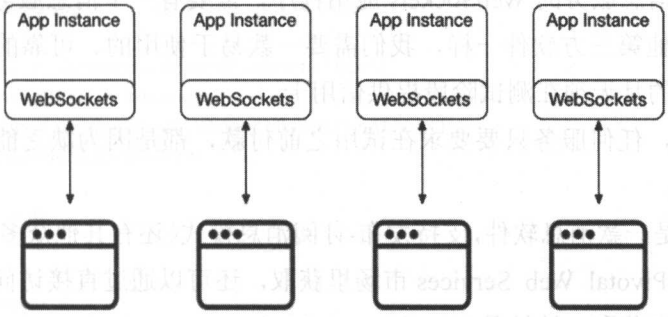

图 11.1 WebSockets 和糟糕的水平扩展

无论使用 SSE 还是 WebSockets，无论是使用我们自己的 WebSockets 服务器实现还是使用框架提供的 WebSockets，此处都有一个问题悬而未决：我们创建了一个个孤岛式的应用程序示例，数据和消息通知却无法跨示例传递。

如果想要让任何连接到我们服务器的浏览器都能够互相交换消息，而不管应用程序示例的数量和位置，那就必须将 WebSockets 组件从示例中抽离，并将它隐藏在后端。我们可以自己实现或者直接使用云端提供的各种消息服务。

这种架构如图 11.2 所示。

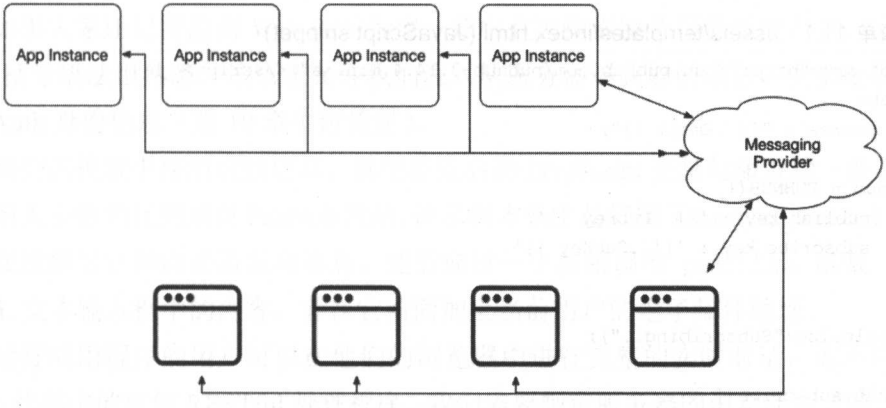

图 11.2 可扩展的、适合云的 WebSockets 架构

在下一节中，我们将看到实现如图 11.2 所示架构的示例代码。

使用消息服务创建 WebSockets 应用

创建基于消息服务的 WebSockets 应用程序，需要有一个消息服务程序。正如本书中使用的其他第三方软件一样，我们需要一款易于使用的、可靠的、支持良好的软件，最重要的是无须在测试阶段提供信用卡。

我们认为，任何服务只要要求在试用之前付款，都是因为缺乏能够让人感到物超所值的信心。

PubNub 是一款消息软件，支持发布-订阅消息模式（还有其他众多功能）。PubNub 服务也可以从 Pivotal Web Services 市场里获取，还可以通过直接访问官网并注册免费账户的方式来获取，地址是 http://www.pubnub.com。

如果大家喜欢 PubNub 的发布-订阅消息的模式（PubNub 中最简单的模式），那么可能也会喜欢它的一些其他功能，比如在线统计和数据流。

使用发布-订阅系统的一个主要优点是，能够根据目标或受众划分消息流量。例如，如果正在构建多人游戏，并且服务器不需要查看聊天消息的内容，那么便可以使用消息中间件实现整个功能，微服务不需要关注聊天流量，只专注于重要的游戏消息即可。

如果想让网页与消息系统通信，可以使用代码清单 11.1 所示的代码同时订阅和发布通道。

代码清单 11.1 assets/templates/index.html (JavaScript snippet)

```
<script src="http://cdn.pubnub.com/pubnub-3.14.4.min.js"></script>
<script>
  var source = "{{ .Email }}";

  pubnub = PUBNUB({
      publish_key : '{{ .PubKey }}',
      subscribe_key : '{{ .SubKey }}'
  })

  console.log("Subscribing..");

  pubnub.subscribe({
      channel : "hello_world",
```

```
      message : function (message, envelope, channelOrGroup, time, channel) {
        console.log(
        "Message Received." + "\n" +
        "Channel or Group: " + JSON.stringify(channelOrGroup) + "\n" +
        "Channel: " + JSON.stringify(channel) + "\n" +
        "Message: " + JSON.stringify(message) + "\n" +
        "Time: " + time + "\n" +
        "Raw Envelope: " + JSON.stringify(envelope)
      );
      var newDiv = document.createElement('div')
      newDiv.innerHTML = message
      var oldDiv = document.getElementById('chatLog')
      oldDiv.appendChild(newDiv)
    },
      connect: pub
  })

  function pub(txt) {
     console.log("About to publish: " + txt);
     pubnub.publish({
          channel : "hello_world",
          message : "[" + source + "]: " + txt,
          callback: function(m){ console.log(m);}
     })
  }

function publish() {
  console.log("Going to publish on demand")
  txt = document.getElementById("theText").value;
  pub(txt)
};
```

如果大家还记得前面 Web 应用程序一章中有关探索服务端模版的内容，应该对{{}}括号语法很熟悉。该语法允许我们注入从服务器端获取的信息，例如配置信息或 OAuth 身份信息（第 10 章中讨论过）。

我们在模版中使用这些信息，以便渲染后的 JavaScript 变量与服务端一致。本示例中绝大多数的代码来自 PubNub 网站。该示例本质上是订阅了 `hello_world` 通道，并且在连接后立即向通道发布消息。随后通过一个按钮调用 `publish` 函数，发送 HTML 文本输入框中的内容，并在它前面加上当前用户的电子邮件地址。

运行应用程序的用户可以在他们的浏览器中进行完整的实时对话，而不需要调用 Go 代码中的任何 RESTful 处理程序。我们需要制定服务器的消息路由策略，而无

须关注采用的是何种消息提供商和技术。

如果希望服务器也能够在这个通道上发送消息，并且这些消息能够显示在已连线用户的浏览器上，那么需要编写一个如代码清单 11.2 所示的处理程序。同样，这其中大部分的代码来自于 PubNub 网站中的示例。

代码清单 11.2 server/broadcast_handler.go

```
package server

import (
        "fmt"
        "net/http"

        "github.com/pubnub/go/messaging"
        "github.com/unrolled/render")
}

func broadcastHandler(messagingConfig *pubsubConfig) http.HandlerFunc {
  formatter := render.New(render.Options{
        IndentJSON: true,
  })

  pubnub := messaging.NewPubnub(messagingConfig.PublishKey,
    messagingConfig.SubscribeKey, "", "", false, "")
  channel := "hello_world"

  return func(w http.ResponseWriter, r *http.Request) {
        successChannel := make(chan []byte)
        errorChannel := make(chan []byte)
        go pubnub.Publish(channel, "[SYSTEM] Broadcast from the server!",
          successChannel, errorChannel)

        select {
        case response := <-successChannel:
                fmt.Printf("pubnub publish response: %s\n", string(response))
        case err := <-errorChannel:
                fmt.Printf("pubnub publish error: %s\n", string(err))
        case <-messaging.Timeout():
                fmt.Println("pubnub publish() timeout")
        }
        formatter.JSON(w, http.StatusOK, nil)
  }
}
```

代码中采用了 Go 提供的 channel 和 goroutine。我们使用 go 语句异步地将消息

推送到 hello_world 通道中。然后使用 select 语句进行阻塞并等待消息到达成功、失败或超时通道。无论哪个通道先获取消息，都将取消阻塞 select 语句并执行相应的 case 语句。

强烈建议大家认真查看 GitHub 存储库中的代码（https://github.com/cloudnativego/websockets）。该代码在开始部分包含了安全认证，因此在浏览器中访问/chat 页面之前，需要使用 OAuth 进行身份认证。

关于 JavaScript 框架

大家可能已经注意到，HTML 页面中编写的用于与消息程序通信并操作页面 DOM 的代码有些原始和简陋。实际上，我们是故意这样设计的。

我们想向大家展示与消息程序进行交互的基本框架，而不是任何特定 UI 框架或庞大笨重的工具链。为此，只有向大家提供简陋的代码，才能激励大家去思考如何让代码变得更优雅（具体见第 12 章和第 13 章）。

另外，访问 PubNub 的网站可以获取更多关于其他功能的示例和文档。基于云的消息中间件为网站提供了一定程度上的实时可编程性，这在几年前几乎是不可能的。在设计和构建响应式应用程序时，忽视这样的服务是很愚蠢的。

运行 WebSockets 示例

要运行 WebSockets 示例，需要从 GitHub 中 pull 最新版本的代码。运行 glide install 确保在 vendor 目录下安装了 Glide lock 文件中定义的依赖（这些依赖没有 check 到 GitHub 的 repo 中）。

接下来，就像上一章的 secureweb 示例一样，确保 local_config 目录中有一个环境变量配置文件。在这个文件中，除了 Auth0 凭证，还需要具有 PubNub 账户的发布和订阅密钥。

如果还没有 PubNub 账户，可以访问 https://www.pubnub.com/并立即创建。

现在可以运行./runlocal local_config/env 了。该命令在 Wercker 构建里启动 Docker 镜像中的 Web 服务器。

现在可以打开浏览器访问 Docker machine 的 IP 地址（如 192.168.99.100），像在上一章中一样完成 OAuth 登录。当界面跳转到用户信息页时，将 URL 更改为/chat，这样就能够参与到如图 11.3 所示的会话中了。

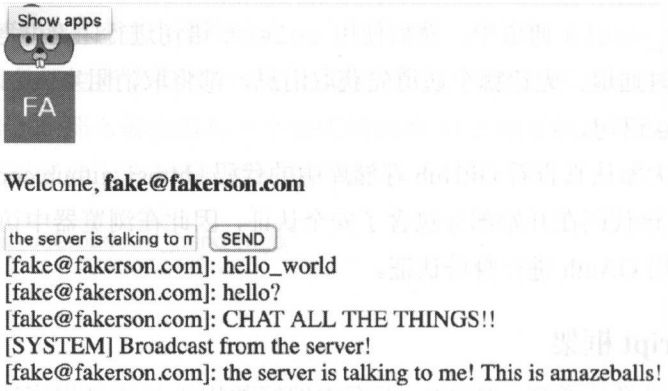

图 11.3 聊天会话示例

另外，可以向/broadcast 发送 POST 请求，触发 Go 代码向所有聊天客户端发送广播消息。

```
$ curl -X POST http://192.168.99.100/broadcast
```

图中的第一个头像是本书的 logo（一只在云端的地鼠）。第二个头像是由 Auth0 创建的，因为用户（***fake@fakerson.com***）没有已存储的图像或已注册的 Gravatar。

在示例输出中，可以看到浏览器会获得用户提交到聊天框的推送通知，服务器进程中运行的 Go 代码会将发送的信息作为系统广播发送到 PubNub 通道中。

作为练习，请大家将代码部署到服务器中，那样就可以允许多个人同时访问它（最好是云，如 PWS），重点观察它在多用户和多浏览器打开时会如何运行，以及同一用户在同一浏览器中打开多个标签页时又会如何运行。

本章小结

能够根据我们现有的 CQRS 和事件溯源模式，构建一个 Web 应用程序用来接收来自服务器的推送通知并做出响应，这一点非常重要。

通过本章，我们了解了为何使用第三方消息系统提供商来实现云原生 Web 套接字，以及它与传统 WebSockets 之间的区别。也了解了如何将第三方消息系统集成到 Go 服务和运行在浏览器端的 JavaScript 中。

12

使用 React 构建 Web 视图

Java 之于 JavaScript 类似 Car 之于 Carpet，根本毫不相干。

——Chris Heilmann，Web 布道师

我们总是竭尽全力地构建微服务，创造一个极其强大、具有扩展性的后台，但绝大多数人的目标远不止于此，有时还需要与应用程序进行交互。

使用浏览器（或移动设备）与应用程序进行交互，基本上都会用到 JavaScript。JavaScript 是 Web 应用程序中不可忽略的一环。

在本章中，我们将介绍 JavaScript 的背景和部分知识点，以及它目前的发展形势和未来的发展方向。在本书中，我们将选择 React 这个 JavaScript 框架，并使用它构建一个简单的单页 Web 应用程序。本书剩余部分的内容和示例中都会用到这些知识和经验。

我们的目标不是使大家成为 React 专家，或教大家如何使用 React，而是教大家如何将 React 应用程序与到目前为止已掌握的 Go 语言技能集成起来，以便为完成最后一章中的示例做好准备。

本章将介绍以下几个方面。

- 了解 React 的基本原理，以及选择它的原因。
- 了解 React 应用程序的结构。
- 构建一个基本的 React 应用程序。
- 介绍如何测试 React 应用程序。
- 更多关于学习 React 的建议。

JavaScript 的形势

长久以来，我们一直躲在优雅、简洁的 Go 生态圈的城墙背后，一旦踏入 JavaScript 生态圈的泥沼（或许用群魔乱舞来形容更为贴切）之后，就必须放下高高在上的心态，将对消费第三方库、简洁优雅的代码、可管理依赖、简单性这几方面的强硬观点搁置起来。

总之，JavaScript 就是一团糟。

如果在 Google 上搜索“如何解决 JavaScript 中的某问题”，肯定会被数以百万计的答案所淹没。问题是，答案没有对错之分。事实上，有无数的框架和库可以用来解决我们提出的几乎所有关于前端、Web UI 的问题。

如果喜爱 MVC 模式，并且只是想构建单页面应用，那么没有比 AngularJS 更合适的框架了。当然也可以使用 Backbone、Cappuccino、Ember、Meteor、Knockout、SproutCore、React 等其他框架。可能到本书出版时，又会有数以百计的框架出现，刚才提到的那些框架又会很快过时。

如果希望拥有很多优秀的组件库用来实现单纯的 GUI，那么可以选择 AngularJS，或者 Bootstrap、CycleJS、ExtJS、jQuery UI 等。

JavaScript 就像是漂浮着无数框架碎片的汪洋大海。构建 Web 应用程序时，最困难的任务不是编写代码，而是决定采用何种框架，以及采用该框架的态度是否坚决。如何在所有漂浮的碎片中找到珍宝呢？

这个问题没有统一的答案。在下一节中，我们将阐明我们的决定，以及做出这个决定的原因。

为什么选择 React

我们必然会构建自己的用户界面，所以决定用 React（https://facebook.github.io/react/）来实现。React 是一个用于构建用户界面的 JavaScript 库，它不是一个 MVC 系统，也不是一个包罗万象的尝试“将一些抽象化”的库。

在本书中，我们一直公开透明地表达所有观点和决定，以及提出这些观点的原因，对于选择 React 也是如此。然而，当阐述选择 React 的原因时，请记住，大家有权质疑和挑战作者。具有批判性思维是现在没有全部使用 QuickBASIC 语言的唯一原

因。

强烈建议大家研究一下 React 的替代方案，看看能否在不考虑框架本身的前提下，在云端构建大规模 GUI 这一点上和我们达成一致。

另外，请查看 SurviveJS（http://survivejs.com/）的相关内容，可以发现一个很有参考价值的关于 React 和其他框架之间的比较结果，以及一套详细的自学基础教程。

虚拟 DOM

无论使用 jQuery、AngularJS 还是其他框架，或者使用更高级的抽象方法来操作浏览器中的内容，最终都要直接改变浏览器的 DOM（文档对象模型）。

当构建现代的、交互式的、可异步在用户和服务器间通信的应用程序时，经常会花费大量的时间建立保护机制，以防 DOM 被相互竞争地使用。如果没有经历过因此造成的痛苦，那么大家应该为此感到庆幸。

直接操作 DOM 也是非常低效的，尤其是在频繁更改时。如果基于大量输入数据对 UI 进行细微的更改，只能以编写代码的方式批量进行改动和聚合，然后定期更新。实际上，我们最终会在应用程序中建立帧率模拟器，防止页面抖动、扰乱接口和频繁更新 DOM 带来的性能低下。

虚拟 DOM 不仅解决了这些问题，并且还带来了更多的好处。由于 React 代码操作的是虚拟 DOM，因此 React 可以将更新汇集起来后作用于真正的 DOM 上，这样可以优化性能并避免写竞争。

我们最喜欢的虚拟 DOM 的一个优点是，虚拟 DOM 的概念是可移植的。可以使用 React Native（https://facebook.github.io/react-native/）工具，让 React 代码在 Android 和 iOS 等移动设备上运行。React Native 的代码运行在比底层设备更高的一个层面上，使用虚拟 DOM 时，更新作用于移动设备的 GUI 而不是浏览器 DOM 上。

> **注意**
>
> 在最初评估 React 时遇到了一个最令人沮丧的问题，就是当创建虚拟 DOM 元素时，很容易认为创建的是真正的 DOM 元素，而不是虚拟的。我们总是忘记中间有一个抽象层，并且疑惑于为何不能直接传递 HTML 属性。

组件组合

比起我们测试的其他库，React 可以在最大程度上缩小框架设计与实际实现之间

的差距，具体是通过组件组合来实现的。

当在一个白板上对 GUI 进行粗略的顶层设计时，一般会绘制图框，而不是实际的控件与 UI。这些框里面包含一些小框，小框里面可能又包含更小的框。

假设我们正在建立一个监控僵尸病毒爆发活动的网页，我们可能会大声道："我们需要有一个病毒爆发报告的列表，每个报告都要有一个链接，点击链接可以跳转到包含详细信息和严重性指示的页面上。"

如果使用 React 实现这些组件，过程可能如下。

- 实现一个名为 `OutbreakReports` 的组件，渲染 `OutbreakReport` 组件列表。
- 实现一个名为 `SourceStation` 组件，显示病毒爆发报告的来源。
- 实现一个 `SeverityIndicator` 组件，以图形方式显示爆发的严重程度（这一部分最有意思）。

完成 React 的学习曲线后，我们将会对这种定义和组合组件的方式感到很自然。平心而论，React 是具有学习曲线的，尽管这听起来很吓人，尤其是对主要从事服务器端开发的人来说，但 React 绝对是一个易于学习和掌握的框架。这对 React 来说是个优势，对于 JavaScript UI 框架社区来说却是一件悲哀的事。

响应式数据流

React 使用所谓的**单向响应式数据流**，这意味着我们构建的所有东西都是为了响应传入的状态变化。我们之所以喜爱 React，是因为它在 UI 层面包含了事件溯源和 CQRS。如果还记得第 8 章的内容，一定会想起我们花费大量的时间讨论的当心态从"一切都是可变的"转变为"所有变化都是作为对不可变消息的反应"时所获得的可扩展性、灵活性和性能优势。

下一章谈到 Flux 时，我们会介绍这个概念的运用方法，它可以帮助我们实现类似 Facebook 这种网站的规模和性能，它每秒可以处理千万级的请求，为百万级用户提供极具响应和互动式的交互体验。

集中焦点

React 并没有试图抽象或封装一切，它只是不想涉及服务器间的网络通信、持久化机制或其他无须关注的东西。

相反地，React 有一个独特的聚焦点：UI。UI 旨在提供一个框架，用于构建与虚拟 DOM 交互的组合式组件。这一点令我们非常欣喜，它能达成简单性的目的，并使这类框架更容易使用。

使用的便利性

我们可以坐在角落观察开发人员构建他们的第一个 React 应用程序了。当疯狂的科学家和超级大反派们惊呼："你说我的代码包有 35000 行代码，你是什么意思？！"时，我们往往会忍俊不禁。

React 实际使用起来相对容易。它很简单直接，不试图完成多件事情。即便如此，它也是一个 JavaScript 库。这意味着我们必须使用 `npm` 和 `webpack` 这类工具，接纳处理一些不常遇到的混乱状况。

如果将 Go 的世界比作一片清爽、干净、优雅、整洁的工作区，那么 JavaScript 世界则是一张杂乱的桌子，上面摆满了溢满咖啡的杯子，到处都是污渍，你的衣服上充满了无法消散的腐败的气味。

这种体验对于一般的 Go 开发者来说是很不愉快的，但我们认为 React 是性能最好、最优雅和简单的框架之一，它是所有云端应用程序 UI 层的理想选择。一旦克服了最初陡峭的学习曲线，挫败感会随之消失，在使用 React（特别是 Flux）时会感到非常轻松愉快。

无论大家相信与否，我们测试的其他库实际上比这个更加混乱和糟糕。这其中的一些框架会在未来几年里始终给开发者带来噩梦般的体验。

React 应用程序剖析

每个 React 应用程序都有一些难解的部分，所有部分都必须很好地组合在一起才能正常运行。像任何优秀的 JavaScript 框架一样，如果这其中有一部分没有正常工作，那么所有东西都将以惊人的方式"炸裂"。

下一节将简要介绍构建 React 应用程序时可能遇到的问题。

`package.json` 文件

`package.json` 是一个项目文件，其中包含了描述 package 的元数据，如果打

算发布该 package，并允许人们通过 npm 来使用，那么该文件将非常重要。此文件中还通过使用 npm 的命令，如 build、start（我们更倾向于使用 watch 而不是 start）或 test 定义了一个 scripts。

代码清单 12.1 显示了将要构建的 package.json 示例文件。

代码清单 12.1 package.json

```
{
  "name": "react-zombieoutbreak",
  "version": "1.0.0",
  "description": "A Zombie Outbreak monitor app written in React.",
  "repository": {
    "type": "git",
    "url": "https://github.com/cloudnativego/react-zombieoutbreak.git"
  },
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no tests\" && exit 1",
    "build": "webpack",
    "watch": "webpack-dev-server"
  },
  "keywords": [],
   "author": "",
  "license": "ISC",
  "devDependencies": {
    "babel-core": "^6.7.2",
    "babel-loader": "^6.2.4",
    "babel-plugin-array-includes": "^2.0.3",
    "babel-plugin-transform-class-properties": "^6.6.0",
    "babel-plugin-transform-object-assign": "^6.5.0",
    "babel-plugin-transform-object-rest-spread": "^6.6.5",
    "babel-preset-es2015": "^6.6.0",
    "babel-preset-react": "^6.5.0",
    "babel-preset-react-hmre": "^1.1.1",
    "css-loader": "^0.23.1",
    "style-loader": "^0.13.0",
    "webpack": "^1.12.14",
    "webpack-dev-server": "^1.14.1",
    "webpack-merge": "^0.8.3"
  },
  "dependencies": {
    "react": "^0.14.7",
    "react-dom": "^0.14.7"
  }
}
```

Webpack.config.js 文件

Webpack（相对应的是 *Babel*）负责在服务器端运行所有的 JavaScript 并生成一个可以在浏览器中使用的单个 *bundle* 文件。Webpack 有很多配置，为了不给大家造成困扰，此处不一一列举。

有关此文件内容的更多信息，请参阅 SurviveJS（本章前面提到过）或其他的 React 在线参考文档。

.babelrc 文件

Babel 工具可以将浏览器不可解的 JavaScript 转换成浏览器可以解析的代码。这里有很多不同的选择，允许我们在服务器上运行比浏览器端版本更新的 JavaScript。

可以在.babelrc 文件中配置要使用的其他插件和解释器。以下是本章中创建的项目示例。

```
{
 "presets": [
  "es2015",
  "react"
 ],
 "plugins": [
  "transform-object-rest-spread",
  "transform-class-properties",
  "transform-object-assign",
  "array-includes"
 ],
 "env": {
  "watch": {
   "presets": [
    "react-hmre"
   ]
  }
 }
}
```

在此文件中可以看到，React 和所有预处理 JavaScript 都遵循 ES2015 标准，同时还使用了一些相当标准的插件。

理解 JSX 和 Webpack

应用程序目录中的所有 JavaScript 文件的唯一使命就是为了生成 bundle 文件。

bundle 是一个经过优化的 JavaScript 文件，可以很好地运行于浏览器中。大家将在本章中看到，我们编写的 JavaScript 代码并不能直接兼容浏览器中的 JavaScript（这是一件好事）。

本质上，我们将 JSX 文件（可以包含嵌入式虚拟 DOM 标记的 JavaScript 文件）和其他 JavaScript 文件编译成 **bundle** 单元，从而隔离了对浏览器中的 JavaScript 引擎的依赖。

React 组件

React 真正神奇的地方在于它的组件。JSX 文件经过 `webpack` 的处理后，由 `babel` 进行转译，最终在 bundle 文件中生成可运行的 JavaScript 代码。

React 组件允许将页面的 UI 划分成逻辑块，正如前面提到的，这通常与将页面架构白板化的方式类似。

按照惯例，我们通常从一个最外层的叫作 `App` 的组件着手。该组件将自身转换为根`<div/>`元素。组件组合和生成的 React 代码 bundle 用于渲染视图页面并处理和用户间的交互。

例如，我们可以创建一个 `App` 根组件，在其中包含一个 `GameBoard` 组件，`GameBoard` 中又存在一个 `Background` 组件和一些 `GamePiece` 组件。

这种组合使得构建强大的用户接口变得相当简单，并且也易于维护和纠错。

构建简单的 React 应用程序

僵尸大灾难即将来临，到处都有病毒爆发，死亡阴影时刻笼罩着我们。这时我们能做什么？何不写一个 App 呢！

尽管我们想完全规避使用样板文件，但事实上这并不可能，特别是在构建 JavaScript 应用程序时。建议大家从 GitHub（https://github.com/cloudnativego/react-zombieoutbreak）或存储库的其他模板中复制完整的 React 应用程序示例代码，而不是自己设置每个项目的框架。

上一节主要解析了项目的样板和框架，接下来我们还需要定义组件模型。在应用程序中，我们会创建一个 `App` 根组件，在它内部创建一个 `Outbreaks` 组件，用来呈现 `OutbreakReport` 组件列表。

自下而上地审视代码，首先检查最低级别的组件，然后逐渐向上，直到应用程序的根。

如代码清单 12.2 所示，OutbreakReport 组件有两种渲染模式：编辑模式和默认模式。开启编辑模式时，可以编辑病毒爆发报告的内容。请注意，为了保持代码简单易读，我们故意禁用了编辑报告中其他字段的功能。

代码清单 12.2 OutbreakReport.jsx

```
import React from 'react';

export default class OutbreakReport extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      editing: false
    };
  }

  render() {
    if (this.state.editing) {
      return this.renderEdit();
    }

    return this.renderOutbreakReport();
  }

  renderEdit = () => {
    return (
        <div>
          <div className="date">{this.props.outbreak.date}</div>
          <div className="origin">{this.props.outbreak.origin}</div>
          <div><br/></div>
          <div className="severity">{this.props.outbreak.severity}</div>
          <div className="description">
            <input type="text"
                ref={
                    (e) => e ? e.selectionStart =
                      this.props.outbreak.description.length : null
                }
                autoFocus={true}
                defaultValue={this.props.outbreak.description}
                onBlur={this.finishEdit}
                onKeyPress={this.checkEnter} />
          </div>
```

```
      </div>
    );
  }

  edit = () => {
    this.setState({
      editing: true
    });
  };
  checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
  };

  finishEdit = (e) => {
    const value = e.target.value;
    if(this.props.onEdit) {
      this.props.onEdit(value);

      this.setState({
        editing: false
      });
    }
  }

  renderOutbreakReport = () => {
    const onDelete = this.props.onDelete;

    return (
      <div>
        <div className="date">{this.props.outbreak.date}</div>
        <div className="origin">{this.props.outbreak.origin}</div>
        <div><br/></div>
        <div className="severity">{this.props.outbreak.severity}</div>
        <div className="description"
          onClick={this.edit}>{this.props.outbreak.description}</div>
        {onDelete ? this.renderDelete() : null }
      </div>
    );
  }

  renderDelete = () => {
    return <button
      className="delete-outbreak" onClick={this.props.onDelete}>X</button>;
  }
}
```

renderEdit 方法处于编辑状态下时会渲染组件，该组件中包含了一个标准的 HTML 输入标签，其中的有些属性可以监听事件并调用事件中相应的方法。renderOutbreakReport 方法用来渲染组件的非编辑状态。

如果单击爆发报告的描述，组件会切换到编辑模式，并提供一个输入框。按下 enter 键，使用 tab 切换或退出输入控件，组件将返回到只读状态。

代码清单 12.3 显示了 Outbreaks 组件的代码，它负责渲染爆发报告的列表。与单个报告组件相比，这段代码更加严密和紧凑。

代码清单 12.3 Outbreaks.jsx

```
import React from 'react';
import OutbreakReport from './OutbreakReport.jsx';

export default ({outbreaks, onEdit, onDelete}) => {
  return (
    <ul className="outbreaks">{outbreaks.map(outbreak =>
     <li key={outbreak.id} className="outbreak">
       <OutbreakReport outbreak={outbreak}
           onEdit={onEdit.bind(null, outbreak.id)}
           onDelete={onDelete.bind(null, outbreak.id)}
       />
     </li>)}
    </ul>
  );
}
```

如果 onEdit.bind 和 onDelete.bind 的语法让大家感到困惑，请不用担心。这些语法对于 JavaScript 开发人员来说是很熟悉的，但对后端和服务开发人员来说可能显得非常陌生。

App 组件将一个方法引用传递给此列表中的每个爆发报告组件，这会在 App 组件中保留事件处理类型函数。下一章讨论 Flux 时可以看到，这种做法比较丑陋，不会在实际生产中使用。

最后来看一下代码清单 12.4 中 App 组件的源代码。

代码清单 12.1 App.jsx

```
import React from 'react';
import uuid from 'node-uuid';
import Outbreaks from "./Outbreaks"
```

```
export default class App extends React.Component {

  constructor(props) {
    super(props);
    this.state = {
      outbreaks: [
        {
          id: uuid.v4(),
          origin: "Station Gamma",
          severity: "Yellow",
          description: "This was bad",
          date: "03-04-2016 12:00"
        }
      ]
    }
  }

  render() {
    const outbreaks = this.state.outbreaks;

    return (
      <div>
        <button onClick={this.addOutbreak}>New Report</button>
        <Outbreaks outbreaks={outbreaks} onEdit={this.editOutbreak}
            onDelete={this.deleteOutbreak}/>
      </div>
  );
}

deleteOutbreak = (id, e) => {
 e.stopPropagation();
 this.setState({
   outbreaks: this.state.outbreaks.filter(outbreak => outbreak.id !== id)
 });
};

editOutbreak = (id, description) => {
  if(!description.trim()) {
    return;
  }
  const outbreaks = this.state.outbreaks.map(outbreak => {
    if(outbreak.id === id && outbreak) {
      outbreak.description = description;
    }
    return outbreak;
  });
  this.setState({outbreaks});
```

```
};

addOutbreak = () => {
   var d = new Date();
   var datestring = ("0" + d.getDate()).slice(-2) + "-" +
       ("0"+(d.getMonth()+1)).slice(-2) + "-" +
       d.getFullYear() + " " + ("0" + d.getHours()).slice(-2) +
      ":" + ("0" + d.getMinutes()).slice(-2);

   this.setState({
      outbreaks: this.state.outbreaks.concat([{
        id: uuid.v4(),
        origin: 'Alpha Fortress',
        severity: 'Green',
        date: datestring,
        description: 'New Report'
       }])
     });
   };
 }
```

在以上程序中，editOutbreak、addOutbreak 和 deleteOutbreak 处理器实现了一般单页 JavaScript 应用程序所期望的 CRUD 功能。这对大家来说可能有些别扭：所有的代码都存在于根组件中，而不是存在于控制器中。

这绝对是一个很正常的现象：混蛋的传道者们（在此指信口开河的作者）在提供优雅的解决方案之前，总会先教给大家困难的实现方式。希望大家理解本书中介绍的 React 应用程序的工作方式，在继续学习后面的内容之前使用它，这将有助于各位在下一章中更好地理解 Flux 的内容。

一切准备就绪（已经从 GitHub 上获取了示例）后，输入下面的命令，确保所有的依赖都安装到了 node_modules 子目录中。

```
npm install -i
```

以上命令会遍历所有的项目文件，并把需要的所有依赖安装到正确的位置。完成这些步骤后，可以在实时更新的 Web 服务器中运行应用程序，这样修改 React 示例也会很容易。

```
npm run watch
```

现在应该可以看到如下输出。

```
$ npm run watch
```

```
> react-zombieoutbreak@1.0.0 watch /Users/khoffman/Code/Go/src/github.com/
cloudnativego/react-zombieoutbreak
> webpack-dev-server
http://localhost:8080/ webpack result is served from /
content is served from /Users/khoffman/Code/Go/src/github.com/cloudnativego/react-
zombieoutbreak/assets
404s will fallback to /index.html

webpack: bundle is now VALID.
```

在 Web 浏览器中打开输出中提供的地址可以访问应用程序。观察应用程序启动时的爆发报告示例，然后添加和删除报告，单击报告描述以进行修改。

在对应用程序的实时更新版本进行了一系列修改后，我们可以构建一个稳定版本，并将它部署到云端。为此，需要使用以下命令生成 `bundle.js` 文件。

```
$ npm run build

> react-zombieoutbreak@1.0.0 build /Users/khoffman/Code/Go/src/github.com/
cloudnativego/react-zombieoutbreak
> webpack

Hash: f0b5b5e1d84529a3067a
Version: webpack 1.12.14
Time: 1636ms
    Asset Size Chunks     Chunk Names
bundle.js 813 kB 0 [emitted] app
    + 189 hidden modules
```

现在 assets 目录下应该生成了一个 `bundle.js` 文件。有了该文件就可以创建 Go 应用程序（包含在示例中和模板中）并运行了。

```
$ go build
$ ./react-zombieoutbreak
[negroni] listening on :8100
```

请注意，该应用程序的默认端口为 8100，而应用程序的实时更新版本（非 Go 版本）的端口是 8080 。

选择何种方式进行开发完全取决于自己，强烈建议大家使用实时更新 Webpack 服务器来完成所有的 React 开发，否则需要经常停止 Go 服务器，重新生成包，并重启服务器。

图 12.1 展示了一个出色的 UI 示例。我们并不是界面设计师，当然也可以在其中

增加自己的创意。

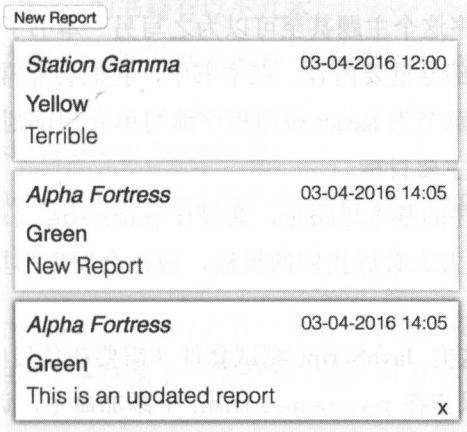

图 12.1 React 应用程序 UI

为了实现鼠标悬停在报告上时能够出现漂亮的阴影和边框的效果，我们从其他地方引用了一些 CSS 代码，除此之外，其他的 UI 都是很基本的。

我们的目标是向大家展示如何将 React UI 关联到用 Go 构建的云服务中，而不是教大家该如何构建丰富的用户界面，有很多专业的书籍是专门介绍这块内容的。

程序运行后，试试对代码的各个方面做些改动，看看 UI 的哪些部分可以自动更新，而哪些需要刷新浏览器才能更新，这个做法绝对是值得的。这期间几乎无须重启 Webpack watcher。

不赞成的做法

本章中创建的示例是经过刻意设计的，它过于简单，没有进行任何网络调用。当复杂度增加时，有些地方就很容易出现问题。

这里存在的最大问题是，在页面层次结构复杂、多个组件位于同一个页面上的情况下，每当有状态变化，所有组件都需要被通知。React 的虚拟 DOM 在状态发生改变时会很好地进行重新渲染，但不会将状态发送给嵌套控件的事件处理器。因此需要一个新的能够保持组件模型简洁性的状态管理抽象层。

我们将在下一章中开始使用 Flux，它可以解决本章中构建 React 应用程序时存在的一些问题。

测试 React 应用程序

测试 React 应用程序这个主题甚至可以为之写另一本书了，至少也应该是单独讲述 React 的书籍中的一部分重要内容。在本书中，我们会非常严格地测试所有的 Go 代码。然而，花一整个章节为 React 应用程序编写单元测试则偏离了使用 Go 构建云原生服务和应用程序的主要目标。

测试 React 应用程序的基本思路是，通过在 `package.json` 文件中添加额外的命令来执行测试。如果大家看过我们的模板，应该会发现那里只有一条信息，显示没有进行测试。

如果使用 Karma 这类 JavaScript 测试套件（需要在项目中有一整套单独的样板文件和依赖项），那么只需在 `package.json` 中添加命令，调用 `karma start` 即可。

与其他任何单元测试一样，单元测试的目标是调用组件中的方法，并断言其相应的状态更改。因此必须深入地了解 React，才能知道如何断言虚拟 DOM 的状态和内容。

进一步阅读

正如本章中多次提到的，我们的目标不是教给大家关于 React 的所有知识，而是为学习下面的几章做铺垫，这需要让大家了解关于 React 的足够多的信息。以下几章将会深入介绍 Flux，然后将本书中使用的所有技术整合在一起，创建一个功能完备的应用程序。

React 网站

当搜索 React 及其他很多替代方案时，我们发现了一本绝佳的参考资料，就是 Juho Vepsäläinen 写的 *SurviveJS*。这本书为那些需要陡峭学习曲线的知识提供了很棒的指南，它从最简单的概念开始，帮助大家一步步理解强大和复杂的概念。其中，Vepsäläinen 也对多种框架表达了自己的观点，并解释了他为何支持那些简单和清晰的框架，这些观点很符合 Go 代码的风格。要浏览或购买此书，请访问 http://survivejs.com/。

React 书籍

我想要推荐的关于 React 的书籍有以下几本。

- *Learning React Native：Building Native Mobile Apps with JavaScript*，Bonnie Eisenman 著。
- *Pro React*，Cassio de Sousa Antonio 著。
- *React: Up and Running：Building Web Applications*，Stoyan Stefanov 著。

其他资料

在 Udemy（http://www.udemy.com）上有一个名为 *Build Web Apps with ReactJS and Flux* 的课程，该课程涵盖了本章及下一章中的主题，并且非常易于理解。

本章小结

本书中有两章内容是讲述 UI 技术的，本章是第一章。虽然本书的重点是使用 Go 来构建云原生应用程序，但我们认为有必要介绍一些关于如何构建高扩展性的用户界面来消费 Go 服务的内容。

虽然不是所有微服务都会被前端调用，但是了解一下如何将目前为止所学的知识应用于实际开发中还是很有用处的。

13 使用 Flux 构建可扩展的 UI

承担错误决定所带来的风险总比因犹豫不决而担惊受怕强得多。

——Moshe ben Maimon （1135–1204 年）

在上一章中，我们初步了解了 React 框架，该 JavaScript 框架仅关注浏览器中的视图部分。它强调采用单向数据流，即页面控件的渲染是为了响应数据状态变更的通知。

本章会更多地关注 UI，这是逐步提升 Go 微服务开发技能的另一个必由之路，不过我们同样会创建一个微服务来支撑可高度扩展的前端。

本章将讨论以下两点。

- Flux 模式及实现方案。
- 使用 Flux 构建一个支持大流量场景的应用程序。

Flux 介绍

和 JavaScript 的某些部分一样，Flux 同样会让人费解。但是当需要构建大规模 Web UI 时，它提供的优秀特性足以让人下定决心挑战它陡峭的学习曲线。如果要实现比本书中的示例还复杂的程序，那就需要花些时间阅读文档或参考其他书籍了。

首先，Flux 不是一个实现，也不是一个库或一组代码，它是一种架构模式。可以把 Flux 当成是 MVC 的替代品，MVC 和 Flux 都是模式，各自有许多不同的实现。

Flux 是基于 React 中引入的单向数据流的概念而构建的，在此基础上添加了许多其他概念，如 **store（数据仓库）**、**source（数据源）**和 **actions（动作）**。Flux 的主要目标是提供一个可行的概念模型，用于将数据和应用程序的状态从视图中分离出来。

前一章中讲到，`OutbreakReport` 组件中到处散布着用于状态查询和操作的代码。此外，我们在顶层的应用程序组件里“塞”进了大量用于状态管理的代码，并且不得不为了管理视图去做各种“丑陋”的绑定和转发。

Flux 通过引入 store 和 action 来解决这个问题。图 13.1 展示了 Flux 架构中各组件之间的数据流关系。

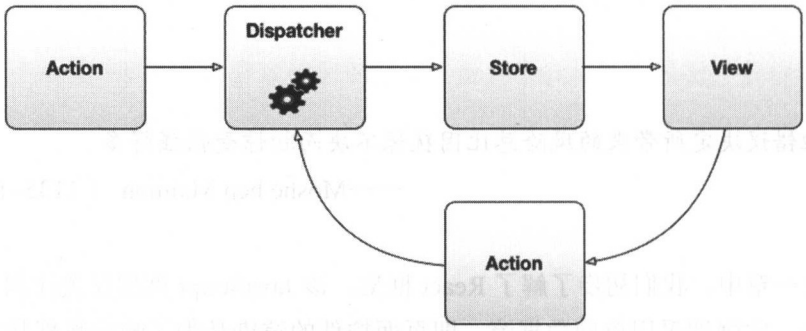

图 13.1 Flux 中的数据流

下一节中将更详细地讨论这些组件。

dispatcher

dispacher（分派器）负责将载荷[译注1]广播到注册的回调函数上。顾名思义，dispatcher 负责集中协调流动于系统中的数据流。

注意上图中的箭头，可以看到 Flux 在基本的 React 架构上添加了更多的概念。不变的是，数据仍然只向着一个方向流动，从 dispatcher 流向 store，再从 store 流向 view。

store

store（数据仓库）是任意数据片段的抽象。如果组件或应用程序需要维护数据或状态，它们就应该被存放在 store 中。也就是说，组件不会像在 React 中那样维护自己的状态。来自 dispatcher 的事件触发 store 中的状态变更，store 中的变更又进一步触发视图的渲染。

译注1 载荷即为后台返回的数据。

有一点需要重申一下，与一些其他双向绑定架构（如 AngularJS）不同的是，视图不直接操作状态，也不能直接与 store 通信。

view

view（视图）是 React 组件，实际上是一个 `render` 函数（即使该函数使用了包装器或抽象）。该函数会在页面中被加载，或在与视图相关的 store 更改时被调用。

action

action（动作）由用户或任何其他因素产生。它们可以表明用户点击一个按钮，或发出其他动作时需要更改相关的状态，例如从 Web socket 或消息提供者处接收消息。

action 的关键之处是，在事件溯源的世界中，许多操作都可以映射到命令（command）中。command 可以是“刷新表格”、“按姓氏排序”、“添加新记录”或“删除记录”等。action 由 dispatcher 处理，然后 dispatcher 广播合适的信息，使 store 可以做出相应的反应。

source

本章将要构建的示例中的一部分并没有体现在图 13.1 中，那就是 source。source 很少在被加载的页面中孤立存在。

例如，我们可以用 store 来维护一个待办事项列表，然后这个待办事项列表由 `Todos` 组件渲染，从而进一步轮流渲染单个 `Todo` 组件。虽然我们会为了示范这个例子而在页面中私有地存放这些数据，但是理想情况下更希望从后台服务获取数据，然后存入 store 中。在这个例子中，这些后台服务是用 Go 实现的。

从示例应用程序中可以看到，我们将响应对应的事件，以便在 store 需要派发数据或响应更改数据的请求时可以与 source 进行通信。

Flux 的复杂性

到目前为止，大家可能困惑于为什么会有这么多的抽象概念。本书中已经花了大量篇幅来阐述选择简洁方案的必要性，但现在似乎正在反其道而行之。

还有另一个规则：更倾向于显式而非隐式。虽然可能有许多开发人员对此有争

议，但作为云上的 Go 微服务开发者，能这样做是很好的。

> **需要遵循的准则**
> 倾向显式而非隐式。

倘若用户界面要具有可以响应来自用户的事件、与 RESTful 服务通信、可以响应来自 WebSockets 的入站事件这些功能，那么以此为标准来构建基于 JavaScript 的用户界面将是非常复杂的。如果非要这样做，结果只能是一团糟。

另外，尽管使用万能的框架（就像是“魔术”）可以让代码量变少，但是当排查故障时，依然不知道问题出在哪里，如何诊断和修复这些问题。最后只会以构建了一个看上去很完美但毫无用处的“巨无霸”而告终。

我们之所以喜欢 Flux 这个模型是因为它虽然有大量的组件，但这些组件都非常小巧且功能单一。此外，使用 Flux 通常不需要自己编写调度程序。正如下面将要构建的示例中所显示的，根据 Flux 的实现，一些样板文件会被隐藏起来。

Flux 提供的是一个单向数据流，位于页面上类似微服务的极小组件里。这意味着在创建应用程序时会生成数量惊人的文件，但是当应用程序出问题或需要添加新功能时，我们也能确切地知道可以从哪里开始排查或添加。

当涉及维护、支持和快速添加功能时，这种架构的显式特性比起其他框架的神奇特性更有价值。

创建 Flux 应用程序

本节将使用前面章节中构建的 React 示例（僵尸启示录），并通过添加三个新概念来简化状态管理和组件设计，这三个新概念是，动作（action）、数据源（source）和数据存储（store）。

回想一下前一个示例中的 `App.jsx` 组件，那是一个非常烦琐的自上而下的设计。我们将事件处理程序植入到顶层组件中，然后将事件不断转发到其他层级的组件中去。我们使用笨重的绑定语法使得代码显得非常臃肿。

如果想查看完整的创建过程，请访问 GitHub 代码库，地址是 https://github.com/cloudnativego/flux-zombieoutbreak。我们已经将 Flux 示例中所有的状态管理代码都放到 **source** 和 **store** 中了，同时也将重要事件的所有通知绑定到 **action** 上了，然后这

些 action 会再被一些相关的组件订阅。

代码清单 13.1 显示了简化过的 App 组件的代码。

代码清单 13.1 App.jsx

```
import React from 'react';
import Outbreaks from "./Outbreaks"
import OutbreakStore from '../stores/OutbreakStore'
import OutbreakActions from '../actions/OutbreakActions'

export default class App extends React.Component {

  constructor(props) {
    super(props);

    this.state = OutbreakStore.getState();
  }

  componentDidMount() {
    OutbreakStore.listen(this.storeChanged);
    OutbreakStore.fetchOutbreak();
  }
  componentWillUnmount() {
    OutbreakStore.unlisten(this.storeChanged);
  }

  storeChanged = (state) => {
    this.setState(state);
  };

  addOutbreak = (event) => {
    OutbreakActions.addOutbreak();
  }

  render() {
    const outbreaks = this.state.outbreaks;

    return (
        <div>
          <button onClick={this.addOutbreak}>New Report</button>
          <Outbreaks outbreaks={outbreaks}/>
        </div>
    );
  }
}
```

在这个组件中，一旦组件被挂载（初始化），storeChanged 处理程序便会监听 OutbreakStore 的更改，然后调用初始提取函数。在这个例子中，初始提取函数检索的是模拟数据，我们也可以很方便地更改 source，让其在 RESTful 服务中检索数据。

花几分钟时间将此代码与上一章中的代码（https://github.com/cloudnativego/react-zombieoutbreak）进行比较可以发现，虽然现在的文件更多了，但它们是分工独立的且更便于阅读和维护。

我们可能不会这样构建一个小应用程序，但当要构建更大更复杂的应用程序（像下一章中构建的那种）时，这种类型的分离是非常重要的。

从代码清单 13.2 中可以看到 action，当通过 dispatcher 发送状态变化时会调用这些方法。action 的返回值最终将作为 store 的参数用于更改状态。

可以将代码清单 13.2 中的内容视为实现本地发布-订阅系统的程序，其中组件是发布者，store 是订阅者，dispatcher 负责派发。

代码清单 13.2 OutbreakActions.js

```
import alt from '../lib/alt';
import uuid from 'node-uuid';

class OutbreakActions {
  updateOutbreak(outbreak) {
     return outbreak;
  }

  fetchOutbreak() {
    return [];
  }

  deleteOutbreak(outbreak) {
    return outbreak;
  }

  changeOutbreak(outbreak) {
    return outbreak;
  }

  outbreakFailed(errorMessage) {
    return errorMessage;
  }
```

```
  addOutbreak() {
    var d = new Date();
    var datestring = ("0" + d.getDate()).slice(-2) + "-" +
        ("0" + (d.getMonth() + 1)).slice(-2) + "-" +
      d.getFullYear() + " " + ("0" + d.getHours()).slice(-2) + ":" +
        ("0" + d.getMinutes()).slice(-2);

    var newOutbreak =
     {
      id: uuid.v4(),
      origin: "Station Gamma",
      severity: "Yellow",
      description: "This was bad",
      date: datestring
     };
     return newOutbreak
    }
}

export default alt.createActions(OutbreakActions);
```

现在，组件不会像之前示例中的那样直接改变自身状态，而是更加符合事件溯源模型。当有事件发生时，action 会发出指令，store 监听这些 action，并通过改变自身内部状态对 action 做出响应。store 内部的状态改变后，只有受影响的组件才会重新渲染。

这些方法仅返回受影响的数据，不会直接操作状态。如果没有 React 的虚拟 DOM 技术这是不可能实现的，在上一章中我们便高度认可过这个概念。

如代码清单 13.3 中的 source，它只是一个 JavaScript 模块，用于连接实际数据与 store。本示例中使用模拟数据，这样可以方便地与 RESTful 服务进行连接。

source 的主要作用是将 *raw* source 数据与用于向组件发送反馈的 store 进行解耦。再次强调一下，所有数据在 React 和 Flux 中都沿单一方向流动。

代码清单 13.3 OutbreakSource.js

```
import uuid from 'node-uuid';
import OutbreakActions from '../actions/OutbreakActions'

var mockData = [
    {
     id: uuid.v4(),
     origin: "Station Gamma",
     severity: "Yellow",
```

```
      description: "This was bad",
      date: "03-04-2016 12:00"
    }
  ];

const OutbreakSource = {
  fetchOutbreak() {
    return {
      remote() {
        return new Promise(function (resolve, reject) {
              resolve(mockData);
            })
      },

      local() {
        return null;
      },

      success: OutbreakActions.updateOutbreak,
      error: OutbreakActions.outbreakFailed,
      loading: OutbreakActions.fetchOutbreak
    }
  }
};

export default OutbreakSource;
```

现在数据已经有了 source，也具有了给组件发送信号的 action，还需要一个 store，即一个页内数据持久化的抽象。store 内状态的变化将自动触发相应 React 的虚拟 DOM 进行重新渲染。

换句话说，我们可以轻易地创建单向绑定，不像 Angular 中那样允许编辑表单来操纵传递给只读层 view 的数据，这里的一切都位于 dispatcher 的下游。此 store 的代码如代码清单 13.4 所示。

代码清单 13.4 OutbreakStore.js

```
import alt from '../lib/alt';
import OutbreakSource from '../sources/OutbreakSource'
import OutbreakActions from '../actions/OutbreakActions';

class OutbreakStore {
  constructor() {
   this.outbreaks = [];
   this.errorMessage = null;
```

```
    this.bindListeners({
      handleUpdateOutbreaks: OutbreakActions.UPDATE_OUTBREAK,
      handleFetchOutbreak: OutbreakActions.FETCH_OUTBREAK,
      handleOutbreakFailed: OutbreakActions.OUTBREAK_FAILED,
      handleChangeOutbreak: OutbreakActions.CHANGE_OUTBREAK,
      handleDeleteOutbreak: OutbreakActions.DELETE_OUTBREAK,
      handleAddOutbreak: OutbreakActions.ADD_OUTBREAK
    });

    this.exportAsync(OutbreakSource)
  }

  handleDeleteOutbreak(outbreak) {
    this.outbreaks = this.outbreaks.filter(target => target.id !== outbreak.id)
  }

  handleAddOutbreak(outbreak) {
    this.outbreaks = this.outbreaks.concat([outbreak]);
  }

  handleChangeOutbreak(outbreak) {
    // make last-minute changes if we need to
  }

  handleUpdateOutbreaks(outbreaks) {
    this.outbreaks = outbreaks;
    this.errorMessage = null;
  }

  handleFetchOutbreak() {
    this.outbreaks = [];
  }

  handleOutbreakFailed(errorMessage) {
    this.errorMessage = errorMessage;
  }
}

export default alt.createStore(OutbreakStore, 'OutbreakStore');
```

调用 `exportAsync` 会将 source 绑定到 action 列表上。在 action 模块中调用 `createActions` 将会生成如 `CHANGE_OUTBREAK` 等由 action 模块暴露出来的常量。

第一次看到这些时，大家可能会对如何将所有这些组件无缝衔接在一起构成一个完整的 Flux 应用程序感到非常疑惑。学习这门技术最好的方式就是全身心投入进

去，改动一些语句，看看程序如何崩溃。

打开 Chrome JavaScript 开发者控制台，更改其中的内容并观察程序是如何运行失败的。沿着 `bundle.js` 文件中看到的失败逆向追溯到由前期的 Webpack 手动更改的 JavaScript 脚本。这样做会得到很多启发，也会帮助我们真正掌握 React、Alt 和 Flux。

`OutbreakReport.jsx` 中的大部分代码是保持不变的。不过，响应编辑和删除不是通过操作控制状态来实现的，而是通过调用 action 来实现的。然后这些 action 通过 dispatcher 进行分派，并以触发绑定在 store 上的 listner 函数作为结束。一旦 store 有所更改，控制器便会重新渲染。我们必须承认单向数据流在支撑复杂 UI 中的重要性。代码清单 13.5 展示了新的 `OutbreakReport` 组件。

代码清单 13.5 OutbreakReport.jsx

```
import React from 'react';
import OutbreakActions from '../actions/OutbreakActions';

export default class OutbreakReport extends React.Component {
  constructor(props) {
    super(props);

    this.state = {
      editing: false
    };
  }

// Master function for rendering, chooses render mode(s).
render() {
  if (this.state.editing) {
    return this.renderEdit();
  }

    return this.renderOutbreakReport();
}

// Renders the control in edit mode.
renderEdit = () => {
  return (
      <div>
          <div className="date">{this.props.outbreak.date}</div>
          <div className="origin">{this.props.outbreak.origin}</div>
          <div><br/></div>
```

```
        <div className="severity">{this.props.outbreak.severity}</div>
        <div className="description">
          <input type="text"
            ref={
                 (e) => e ? e.selectionStart =
            this.props.outbreak.description.length : null
            }
            autoFocus={true}
            defaultValue={this.props.outbreak.description}
            onBlur={this.finishEdit}
            onKeyPress={this.checkEnter}/>
        </div>
      </div>
    );
}

// Switch the control into edit mode. Does NOT invoke a store-changing action.
edit = () => {
  console.log('Switching to edit mode.');
  this.setState({
    editing: true
   });
};

checkEnter = (e) => {
    if(e.key === 'Enter') {
      this.finishEdit(e);
    }
};

// Blurred out of edit control. Commit changes to store via Action.
finishEdit = (e) => {
  console.log('Exited edit')
  const value = e.target.value;
//this.props.onEdit(value);
  this.setState({
    editing: false
  });
  if (!value.trim()) {
   return;
  }
  this.props.outbreak.description = value;
  OutbreakActions.changeOutbreak(this.props.outbreak);
}

// Invoke delete action to modify store.
onDelete = (e) => {
```

```
    console.log('Clicked delete');
    OutbreakActions.deleteOutbreak(this.props.outbreak);
}

// Renders read-only version of report.
renderOutbreakReport = () => {
  const onDelete = this.props.onDelete;

  return (
   <div>
     <div className="date">{this.props.outbreak.date}</div>
     <div className="origin">{this.props.outbreak.origin}</div>
     <div><br/></div>
     <div className="severity">{this.props.outbreak.severity}</div>
     <div className="description"
        onClick={this.edit}>{this.props.outbreak.description}</div>
     {this.renderDelete()}
    </div>
  );
}

renderDelete = () => {
  return <button
    className="delete-outbreak" onClick={this.onDelete}>X</button>;
  }
}
```

我们对该组件的一系列重要变化做出了强调。如上所述，用户与此组件的交互现在已重构为通过触发 action 来实现。action 通过 dispatcher 进行分派，然后被相关的 store 监控。store 的更改会导致相关组件重新渲染。我们已经重复过许多次上面的流程，希望这种重复的数据流能够帮助大家记忆。

为了判断自己是否处于编辑模式，`OutbreakReport` 仍将维持原来的内部状态。我们根据 store 和 action 的概念将这一部分分离出来，之所以用这种方式处理是为了让大家看到同一个示例的两种不同实现方法。可以试着将 `isEditing` 标志作为状态添加到 `OutbreakStore` 的记录中进行实践，看看自己更喜欢哪种模型。

`Alt.js` 就像机器中的齿轮，它提供了一些绑定封装器，这样能够避免直接和 React dispatcher 进行交互。我们看过了很多别人编写的十分丑陋和混乱的示例代码，它们直接在 React 和 Flux 的最底层使用 dispatcher。Alt 不会让人觉得很难看，因为它隐藏了很多的样板文件。

如果查看 GitHub 仓库，会看到一个 `lib/alt.js` 模块创建了一个 Alt 的示例，但是这一章中可能不会出现。

在实际操作中，要先从 GitHub 中获取最新代码（`Git pull`），并输入以下命令。

```
$ npm install
$ npm run watch
```

现在可以像上一章一样打开浏览器并输入 ***localhost:8080***。UI 也与上一章的示例完全相同，但是为了支持更复杂的 UI，代码被分离和重构了。为了满足云中部署的微服务的可扩展性，现在可以开始着手编写 UI 了。

可以通过执行以下命令在 Go 服务器中托管此应用程序。

```
$ go build
$ npm run build
$ ./flux-zombieoutbreak
```

使用 `webpack` 生成 `bundle.js` 文件，然后使用 Go 服务器托管静态 asset 以启动应用程序。

如果不想构建任何东西就直接运行应用程序，那样便可以直接从 Docker Hub 上获得镜像并执行。

```
docker run -p 8100:8100 -e
  WEBROOT=/pipeline/source cloudnativego/flux-zombieoutbreak
```

本章小结

我们既不是 JavaScript 程序员也不是 React、Flux 或 Alt 方面的专家，我们要先承认自己对这方面不是很精通。我们的专长是微服务，即用最受欢迎的 Go 语言来构建微服务。

如果你是一个 React 或 Flux 专家，可能已经注意到前两章中有些示例不是惯用或公认的 JavaScript、React 和 Flux 的最佳实践。但是对于一个新手来说，本书中介绍的 Flux 和 React 技巧也说得过去。

这些章节涉及的所有项目都是开源的，如果有不满意的地方，欢迎随时提交 pull request 帮助我们改进。

这些章节的目的不是教大家 JavaScript、React、Flux 或 Alt 技术，而是说明创建适用于微服务的交互式前端有多复杂。比起本书中为讲解 Go 的云原生编程而涉及的

前端技术来说，有大量的资源和书籍更细致地教授这些技术。

在下一章中，我们将使用本书中介绍的所有技能进行实践，希望能让大家觉得辛苦看完 JavaScript 和 UI 这两章是值得的。

14

创建完整应用 *World of FluxCraft*

记住！独自行动是很危险的！

——Old Man NPC [译注1]，1986 年 2 月 21 日

构建微服务其实是构建一个服务生态系统，使其产生真正的商业价值，让股东和客户满意，不是仅构建一个微小服务就可以的。

微服务生态系统的核心是分布式系统。构建分布式系统、微服务和具有许多可独立部署扩展组件的应用程序是很困难的。

在阅读编程书籍时，我们肯定希望能够运用从众多示例中学到的东西来创建可用于生产环境的应用程序。纵观前面十几章，我们漏掉了一些重要的东西。到现在为止本书只教给大家如何创建单独的例子，而没有教大家如何创建实际可用的应用程序。

本章会运用目前为止学到的所有知识来构建一个大型应用程序。这个应用程序是一系列微服务的集合，它们都是可以被单独部署并自行扩展的。简而言之，我们将运用 1~13 章中学到的所有技能来构建一个大型云原生应用程序。

虽然本章中构建的是一个名为 *World of FluxCraft* 的游戏，但这里讨论的所有技术和架构模式基本都可以应用于其他的领域中。

本章包括以下内容。

译注 1　Old Man NPC（非玩家角色）是任天堂出品的《塞尔达传说》系列游戏中的反面角色，在不同的系列中都以一个白胡子老头的形象出现，他们实际上是不同的角色，可能帮助或阻碍 Link 的冒险。详见 http://zelda.wikia.com/wiki/Old_Man。

- *World of FluxCraft* 游戏背后的设计理念。
- 分布式游戏系统的顶层架构。
- Flux GUI。
- 命令处理器概览。
- 事件处理程序概览。
- **reality server** 概览。
- 地图管理。
- 分布式系统自动化验收测试概览。
- 完整的功能示例代码。

World of FluxCraft 介绍

编写程序设计书籍的最大难点之一就是选择示例代码。人们经常要求作者能够以一种既不会使读者厌烦又不会掩盖其中重要细节的方式来说明如何解决极其复杂的问题，但我们又不能把生产环境应用中的数千行代码都列举在书里。

我们还需要考虑范围。就像本书中构建的应用程序和服务一样，范围蔓延[译注1]（scope creep）在项目管理中是指不受控制的变化或持续增长的项目范围。通常是因为范围欠缺定义、文件化或控制而产生的，而且通常被认为是负面的。对于现实生产环境中的应用程序来说也是一个问题。*World of FluxCraft* 中实现了范围、复杂性和可读性的平衡，我们将本书中的所有技术结合成了一个简单的示例，希望这样不会让读者感到过于复杂。

World of FluxCraft 是一个基于浏览器的多玩家游戏。这是一个为小组玩家创建的非常小巧的游戏，而非 MMORPG（大型多人在线角色扮演游戏）那种长时间沉浸类游戏。

游戏的规则很简单——到达地图上的获胜点。我们将以敏捷方式构建软件，首先要为应用程序确定最小的可用功能（MVP 或最小可行性产品），应用构建完成后就可以在使用核心功能的同时添加和增强功能。

首个玩家选择地图开始游戏后可以邀请其他玩家进入游戏。玩家在地图上四处

译注1　范围蔓延又称为需求蔓延、功能蔓延、特征蔓延。

移动，头像代表他们的位置。玩家之间可以相互聊天和攻击，能够造成随机数量伤害的攻击是多玩家对战游戏中最基本的功能（MVP——最小化可行产品）。

World of FluxCraft 可以作为构建基于浏览器 MMO（大型多人在线游戏）的起点，也可以当成是利用模式和代码来构建交互式实时业务应用程序的开始。本章讨论的是架构思想、高级设计以及微服务生态系统的数据流序列，这其中不包含任何代码。

查看 GitHub（http://github.com/cloudnativego/wof-*）中以 `wof-`为开头的仓库可以获取代码，查看构建 *World of FluxCraft* 应用的方法及将其部署在云中的方法。

图 14.1 是 *Worldof FluxCraft* 游戏地图的早期渲染原型图（我们在网上购买了一些漂亮的图像拼贴和头像，毕竟两个玩家之间不能直接以一条直线相连）。

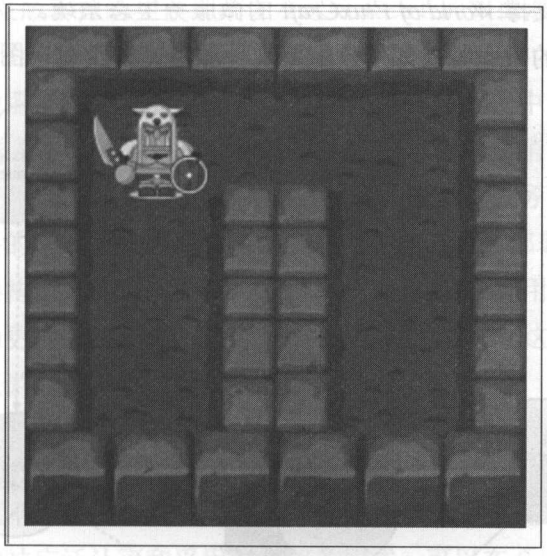

图 14.1 *World of FluxCraft* UI 的早期原型图

架构概览

当我们在玩游戏、查看银行余额或使用社交媒体登录自己喜欢的餐厅网站时，一般只能与系统的冰山一角进行交互。每次这样的交互背后通常都隐藏着大量的代码和基础组件。

在示例游戏中，作为与 *World of FluxCraft* 网站主页进行交互的玩家，通常都希望能直接导航到基于 Flux 的 UI 页面上。

进入游戏后，在游戏界面上移动或与游戏中的其他玩家交互时，移动操作被发送到命令处理器，命令处理器的工作方式与第 8 章（事件溯源和 CQRS）中的一样，命令被转换为事件并发送到消息队列系统。

事件处理器的任务是处理事件。在事件处理过程中，每个事件都被持久化在历史归档中。有些事件可能还需要向实时消息基础组件（如 Websockets）发送消息。我们可以直接向运行中的游戏推送消息。最后，经过计算后的游戏状态会被发送到 reality server 上作为高速缓存。

每场游戏都可以使用不同的地图，因此要具有存储和检索地图元数据（如图块、图形、障碍物）的能力，地图服务如图 14.2 所示。

下图展示的是支撑 *World of FluxCraft* 的微服务生态系统。 为了简洁起见，有些在最终实现中使用的微服务（如用户配置文件服务）并没有在图中显示。

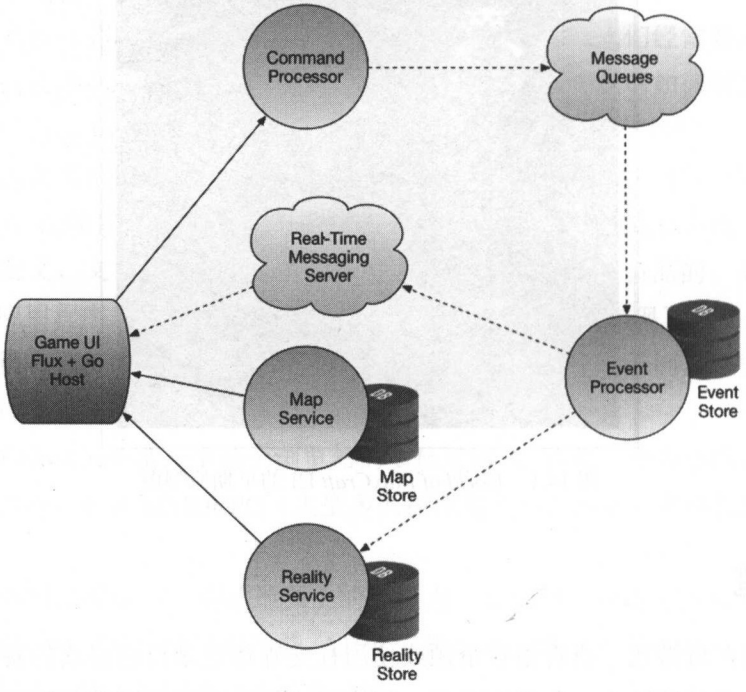

图 14.2　程序上层架构

以下几节将深入地分析每个组件的设计、架构和决策。

独立扩展、版本控制和部署

图 14.2 中的每个圆圈都是一个微服务。本章中的微服务设计可以确保它们能被独立地部署，进行版本控制和扩展。

系统在运行时也可以完成部署。我们可以做一个蓝/绿发布来发布新的微服务，本质上就是可以同时部署老版本和新版本，然后测试新版本，确保一切准备就绪后便可以把旧服务的流量迁移到新服务中去。这种发布很有可能会把事情搞砸，例如发布了一个服务，而这个服务中的 API 却已被更改或破坏。

这个问题就引出了下一个概念：版本控制。实际上可以在系统运行的同时部署新版本还不影响系统运行，但是需要遵守一些基本准则，例如尽可能避免更改公共 API。可以使用语义版本控制类技术，让发送给 REST 的数据或资源 URI（或两者）中包括版本信息。

图中的每个微服务都是无状态的。虽然图中存在持久化中间件，但服务本身是无状态的。这些服务的设计宗旨是能够快速启动并可以快速销毁。这意味着，我们可以根据需求和使用模式实时地按比例扩大或缩小每个微服务的实例数，而不需要停止系统。更重要的是，不必为新加入的实例而更改或重新部署配置。

此外，这种架构还允许自动伸缩。根据系统中的 CPU 利用率、队列深度或其他高级业务指标启动更多服务示例。对于像 *World of FluxCraft* 这样的游戏，可以根据并发用户数或活跃游戏数量来进行扩展。

数据库不是集成层

数据库永远也不应该作为集成层。在图 14.2 中，每个微服务都有自己的私有数据存储。这是一种被广泛认可和采用的微服务模式，可以在 Sam Newman 的书中找到更多这方面的信息。

单向不可变数据流

再看一下图 14.2，数一下其中包含的双向箭头或双向通信的数量，竟然一个也没有。每次与系统交互时，数据都始终沿着单向可预测的方向流动。数据只会从 source 流向目的地而不会向相反的方向流动。

这充分反映了 React 和 Flux 的架构设计原则，也印证了第 8 章中讨论的事件溯源和 CQRS 背后的指导原则。这并不是巧合。

可变性是可扩展性的的克星。系统越多地采用响应式、单向流动模式，其可扩展性就越强。得益于架构的灵活性，我们可以在系统运行后用令人惊奇的方式扩展这些系统，使其获得巨大的性能提升。

Flux GUI

对于大多数玩家来说，*World of FluxCraft* 的用户界面显然不符合当前在线角色扮演游戏的制作水准。年纪大一点的人应该还记得小时候在街头游戏机上玩的 25 美分一局的 8 位视频游戏 Dankey Kong（大金刚）。我们仍记得第一款 8 位电子游戏的界面。

为了向当今数百万开发者的童年致敬，我们决定把 WoF 打造成一个经典的自上而下视角的八位地图爬虫游戏。可以在 https://github.com/cloudnativego/wof-ui 上找到 WoF Flux GUI 的完整代码。

游戏的主界面包括自上而下视角的地图视图，玩家的头像将在地图上移动。玩家只能按规定的方向移动，并且只能以 Flux GUI 和后台服务允许的方式与地图进行交互。

Flux GUI 的大部分任务都在 `MapSource` 和 `MapStore` 组件中完成。我们需要从地图服务中获取地图元数据。当有玩家在地图上移动时，就需要修改存储器来反映这一变化。存储器更新后，所有相关组件（如 `MapTile`）都会被重新渲染，以显示玩家的新位置。

我们发现了一个特别有用的技巧，即可以不借助成熟的 2D 游戏引擎，而是直接使用 **sprite** 来渲染 Flux UI。从传统意义上讲，sprite 是一个 2D 图像，可以围绕屏幕或其某一截面进行移动。

有一个经常被人忽视的 CSS 功能可以带来极大的便利——将 HTML 组件的背景设置为由一个个小矩形组成的大图。这样能够得到一个个表现为单一 PNG 图像的典型八位剪贴图片集。然后设置一个样式表，用来声明位于图形中各个坐标下的剪贴图的类名。

所以，假设有一个包含石头上的下水道格栅背景的瓦片，它在图像内的偏移是 768×384。我们可以很容易地声明一个 CSS 样式，将这个图像作为 div 的背景。

```
.tile-sewer-grate-on-stone {
```

```
  background: url('/tiles/dungeon_tileset_128.png') -768px -384px;
}
```

任何时候声明一个具有这种类名的 div 都会产生一个下水道格栅的背景。如果想要在界面上放置一个玩家头像，或者像杠杆、开关或其他不能通过的障碍物之类的图标，可以在该 div 中放置一个图像标签。想让一个 sprite 在地图上行走，不必费力去做一个真正的 2D 引擎。

下面这个简单的 div 表示 `MapTile` 组件中的地图图块。

```
<div className={className} onClick={tileClicked}>
  <img id={tile.sprite} src="/sprites/img_trans.gif" onClick={spriteClicked}/>
</div>
```

`tile.sprite` 变量字符串对应 sprite 图像的 CSS 样式，可将其映射到 sprite 图层中的子矩形上，就像对背景图块进行绘制一样。在 div 上设置类名将会更改背景图块。

为了支持处于地图拼贴之上的 sprite，需要用一些 CSS 技巧来巧妙地动态布局整个游戏地图。由于地图中使用的是简单的图片，因此需要用一些更巧妙的方式来表示多个玩家当前占用的是同一个地图图块。

创建这个游戏的目的不是为了直接与动辄拥有数百万美元预算的 MMORPG（大型线上多人角色扮演游戏）竞争，而是教大家运用本书中学到的技能来创建运行自己的应用程序。因此，我们制作了最简单的 UI，与其将精力用在花哨的动画、音效、视觉滚动上，还不如增强游戏的可玩性和构建基于 Go 语言的后端服务。

Go UI 宿主服务

玩家发生各种行为，如移动、聊天、加入和退出游戏，都会发送命令到命令服务上。一般不允许 Flux JavaScript 与命令服务直接通信，因为这样会带来一些安全和基础架构层面上的问题，如跨站点脚本执行的安全问题。

这些行为发生后会被 post 到本地服务上，而不是命令服务上。本地服务仅仅是负责获取后台返回的数据并将其发送到命令服务中。

构建更复杂、支持移动设备和 Web 端的游戏应用时，可能会用到 Apigee 这类第三方服务。这些服务提供 API 网关服务，应用的所有流量都通过 API 网关，另外，这类服务具有各种强大的功能，如条件转换、安全性、攻击预防、入侵检测甚至地理优化等。

使用 Go 应用程序作为命令服务的简单代理，承载 Flux UI，演示游戏及其相关服务应该足够了。

玩家移动时序图

图 14.3 中的时序图展示了整个系统中流过的信息流，这些信息流是由玩家移动到地图上的新位置发生的动作而产生的。

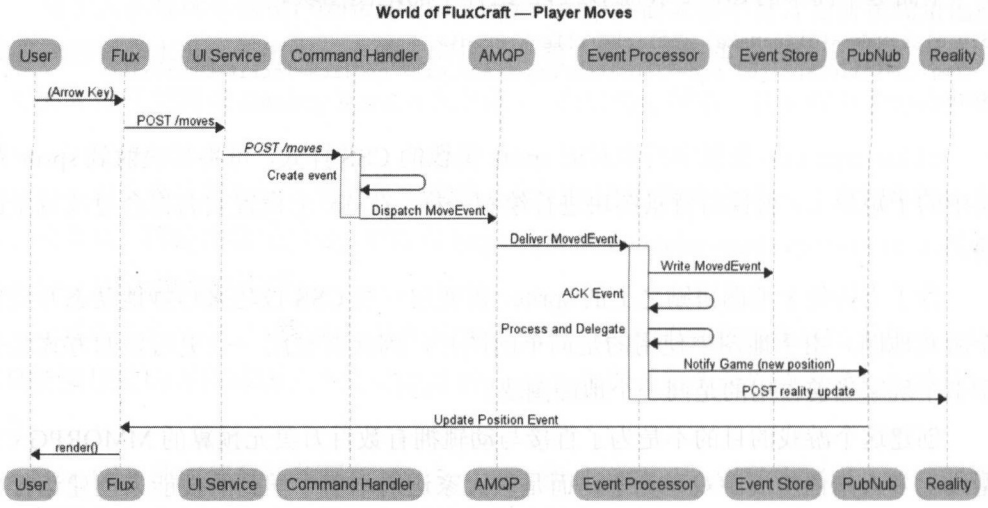

图 14.3　玩家移动的时序图

来源：websequecediagrams.com

图中时序如下。

1. 用户按下光标键在地图上移动。
2. Flux UI 响应按键动作，修改本地地图存储，渲染对应的组件。
3. JavaScript 将移动命令的 JSON 数据（payload）发送到本地 URL `/api/moves` 上。
4. UI 宿主服务将该数据转发到命令处理器服务中。
5. 命令处理器把输入的命令转换成事件并将其分派到队列中。
6. 事件处理器从队列上选取事件后可以进行以下处理。
   a. 将事件写入事件存储器。
   b. 发送通知（如果适用）到实时消息系统。

c. 计算出新的游戏状态并提交到现实服务器中。

我们将在分别构建每个微服务的时候详细地讨论这些步骤。

命令处理

正如第 8 章所说，命令处理器的任务就是处理指令。命令是意图的声明。客户端向系统发出命令时不需要关心命令究竟如何执行，以及执行后将产生什么样的结果。

更重要的是，命令是以异步的方式被发布的。向命令处理器提交命令后会得到一个返回码，该返回码表示发送的数据是否有效、命令是否被成功分派，但处理和实现这些命令可能需要几秒、几分钟甚至几小时，这取决于应用程序的类型。

FluxCraft 中的绝大多数命令应当被近乎实时地处理，或者尽可能被系统以最快的速度处理。这样做既有优点又有缺点。优点是，如果想要让事务被更快处理，那么可以启动更多的处理器，或者把事件分派给更多的消费者。

近实时处理的缺点是，当系统出现卡顿时，玩家会立马感受到。玩家的移动和聊天消息可能会出现延迟等。

World of FluxCraft 命令处理器具有以下职责。

1. 暴露 RESTful API 并通过 POST 请求接受新命令。
2. 将命令转换成事件，使用环境元数据（如时间戳）来扩展事件。
3. 在不知道潜在消费者的情况下分派事件。本书的例子中将新生成的事件分派给 RabbitMQ 队列。

命令处理器的职责是将命令转换成事件，这本质上是把对某件事发生的期望转化成该事件已发生的记录。换句话说，命令是现在时，而事件是过去时。

这其实是一个很小的责任集（可以说是单一的责任），我们是有意而为之的。这就是微服务概念中的“微”。服务的职责越单一，更改该服务的影响就越小，而在给应用程序添加新功能时修改它的可能性就越小。

下面列举了如 *FluxCraft* 这种类型的游戏所支持的命令。

- 创建游戏
- 加入游戏
- 发送文字消息

- 攻击
- 移动
- 离开游戏

这些命令都是由玩家主动发起的。

事件处理

事件处理器是本书中与纯函数式编程思想关系最密切的部分，它的职责是接收和处理事件。处理事件实际上是通过把事件应用于旧状态来计算出新状态的过程。

前面多次提到过，在事件源系统中处理的事件必须是幂等的。意思是将事件应用于已知状态时，保证始终得到完全相同的结果。这个原理不仅可用来构造测试模块，同时也是构建高可靠、可扩展系统的关键。

FluxCraft 的事件处理器从队列中接收到的待处理事件包括以下几个。

- 玩家加入游戏
- 玩家离开游戏
- 玩家发送文本
- 玩家移动
- 玩家发起攻击
- 游戏开始

注意，之所以像上面这样刻意描述事件，是因为要体现出它们发生在过去这一事实。

事件处理器接收到事件后执行相关函数，通过事件来更新旧状态。新计算出的状态发送到 **reality** 服务（稍后讨论）中，而事件本身保存在 event store 中。

如果适用的话，通知将被广播到系统的各相关组件上，通知它们有新状态（或新状态的子集），即事件处理的结果。大家可能会想，发出通知是一个有副作用的表现，所以这不是纯粹的函数式编程。

确实如此，执行函数后发出通知的方式不是纯函数式编程。如果按照百分百纯粹的函数式编程构建这个示例的话，事件处理器应该产生次级事件流，然后处理这些事件。次级事件流会通知和广播各相关方。

为了避免范围蔓延，我们不会采用这种方式，因为事件发出的通知不是为了改

变数据，而是为了方便数据变更的广播。我们选择通过消息子系统（*FluxCraft* 使用 PubNub）来推送新信息（如玩家生命值和位置），这些信息也可以通过查询来获得。

为了确保事件处理器在云中正常运行，也为了方便起见，我们暴露了一个 REST 端点，通过这个端点可以查询事件处理的计数器。在具有更大问题域的复杂系统中，事件处理器上的 REST 端点可能会暴露查询 event store 的方法。

维持现实服务的状态

在第 8 章中，我们说现实是事件溯源的。现实是我们的大脑对多个永无止境的事件（刺激）进行疯狂复杂地计算而得到的最终结果。

World of FluxCraft 中的“现实”同样也是对一系列游戏相关事件进行计算后的结果。任何给定游戏的状态（现实）都可以由事件处理函数以顺序处理游戏事件流的方式来确定。

浏览器首次加载游戏页面时，需要知道正在进行的游戏状态。浏览器没有事件历史记录，也没有事件处理器逻辑，因此需要向现实服务查询游戏状态。

当事件处理器将新事件应用于现有状态时，新状态会更新现实服务，使上面提到的情况（加载游戏玩家网页）下的状态可用。

现实状态服务器的工作非虽然非常简单，但却至关重要。为了避免范围蔓延，本书的示例使用简单的 MongoDB 数据库作为现实服务后台。在现实场景中，可能同时存在数以万计的并发的游戏（现实状态）动作，这时可以升级现实服务，使它与持久性服务之间存在一个缓存层。

与所有其他服务一样，我们可以随时部署此服务的新版本，而不影响系统的其他部分。根据康威定律，这也意味着可以让不同的团队为不同的服务工作，并仍能快速构建和部署新功能。

地图管理

开始新的 *World of FluxCraft* 游戏时需要选择地图。我们脑海中首先浮现的可能是一系列大气的多人游戏地图：*Nuketown*、*Blood Gulch*、*Rainbow Road*、*Gold Rush*、*Goldeneye Archives* 等。

FluxCraft 的地图除不够大气、互动少、缺乏乐趣、没有动感的图形界面、零预算的开发与运营成本、只在浏览器中运行外，与以上的地图没有什么本质的不同。如果不是因为这些，或成百上千的其他原因，我们也能创建出让大家满意的全时段多玩家游戏地图。

有些多玩家游戏地图是由互连的几何多边形、网格、光源以及无数其他复杂图形组成的巨幅图像。我们使用的地图灵感来自于 8 位游戏时代的自顶而下视角游戏的地图。因此，*World of FluxCraft* 的地图是由 2D 网格图块组成的。

地图内的每个图块都具有图像属性（图像由浏览器通过 CSS 提供）和元数据，元数据中包括玩家可否遍历该地图图块、能否占据该空间、使用哪些方向退出图块等。

地图上的每个图块都有唯一的 ID，*World of FluxCraft* 游戏中安装的每个地图也都有自己唯一的 ID。查看 GitHub 中的代码可以看到事件处理器和 UI 如何使用地图和地图图块的唯一 ID。

使用由单独的微服务管理的地图有一个非常酷的作用，就是可以在游戏的主 UI 中利用地图渲染器技术创建自己的**级别编辑器**和**地图设计器**。这可以“逃避”手工构造地图 JSON 数据的艰巨任务。

玩家启动游戏后，UI 宿主服务会向地图服务器发起查询请求获取地图元数据，如果创建了级别编辑器，级别编辑器将向地图服务提交新地图或对现有地图进行更改。

值得一提的是现实（游戏状态）操作地图的副本，该副本使游戏与地图的变更相互隔离，满足了事件源系统的无外部数据需求。如果不得不在事件处理的过程中不断地与地图服务进行交互，那么事件处理就既不可预测也不再是幂等的了。

自动验收测试

即便是这样一个明显降低了难度的多人游戏的例子，其中也还是有很多组件的。一个命令处理器、一个事件处理器、一个现实服务器、一个实时消息组件，还有基于队列的消息和游戏主用户界面。此外还有一个地图服务器和一个用户配置文件服务，如果想继续添加功能，预计微服务数量还要增长。

我们一直秉承云之道的方式以测试先行的方法构建了所有测试，并进行了隔离

测试。但是仅进行隔离测试是不够的，还需要一个可靠的预测器来确保将所有单独的服务都投入生产后能够按预期进行工作。

为此，我们需要进行验收测试。对于多玩家游戏最直接的测试方法是使用游戏脚本。游戏脚本本质上是玩家在游戏中的动作序列。单个游戏的验收测试包括以下内容。

1. 将所有微服务部署到纯净的环境中，清理验收测试的环境配置。
2. 向命令服务器发出启动游戏命令。
3. 发出几个添加玩家的命令。
4. 运行游戏中玩家所熟悉的命令序列。命令序列将尽可能包括所有类型的命令，最好是多次执行一些具有预期结果的场景。
5. 等待片刻，游戏会继续运行。
6. 查询现实服务器，确保游戏的状态符合预期。

这不仅仅是重新测试所有组件，也是在测试它们作为一个整体是否能够按预期工作。不需要把 Flux/React UI 作为验收测试的一部分，因为它们位于 websocket 接收器以及向命令服务发送 POST 命令的发送者之上，仅仅在表现层而已。

完成了各种游戏场景的排列验收测试后，大家可能想要提高验收测试的复杂性。*World of FluxCraft* 毕竟是一款多玩家游戏，除了能够支持多个玩家参与到单个游戏中，还需要能够支持大并发的游戏场景。

为了确保 *FluxCraft* 能按我们所希望的那样正常运行，应该同时开始多个游戏，并同时向这些游戏发出命令。使用 **goroutine** 可以很容易地实现——每个 goroutine 加载一个游戏脚本（可能是一个发送给命令服务的 JSON 数据文件）并发送。

测试结束后，检查现实服务中每个游戏的状态，确保每个游戏的状态不会蔓延或干扰到其他游戏。这种验收测试也适用于向 *FluxCraft* 环境推送配置数据，扩大验收测试的客户端数量和每个测试中的游戏并发数，给服务器增加压力，确保所有游戏都正常运行。

最后，测试中还应包含写有无效移动命令的游戏脚本。我们需要确保如果有人想要篡改与命令服务通信的协议，他们仍然无法移动到禁止的点，攻击够不到的玩家，或以其他方式欺骗或滥用系统。

正如这本书中所做的，每次提交都会触发一次 CI（例如 Wercker）构建，该构建

运行所有单元测试和所有更改后的微服务的集成测试。然后，继续通过 CD 管道启动一个刚更改过的微服务以及所有其他验收测试，确保所有更改不破坏游戏的整体运行。

如果微服务生态系统中没有这样的自动验收测试，那么我们会对该生态系统是否具有能够支持适当游戏的能力产生怀疑。没有玩家抱怨，说明游戏很好。这种模型是运行一个稳定、可扩展、生产级系统的有效方式。

我们把时延和夜间验收测试套件的状态显示到监控上，无论团队成员身在何处都可以立即知晓最近提交的代码是否破坏了游戏。事实上，我们曾在一些地方这样做过，团队已经装备了熔岩灯，愤怒的达斯·维达正盯着验收测试的状态。

显然，验收测试失败时，修复它们就成了优先级最高的事情，因为如果验收测试中断，团队就无法按原进度每天及每周向生产环境 push 代码了。

当然，如果全部按照上面这种方式来构建测试，则可以确保游戏在所有的关键场景中都能正确运行。如果对此不用心，绿色验收测试灯只会带来假的安全感，第一个发现软件问题的人将是客户。

本章小结

World of FluxCraft 虽然拥有不可思议的原创名称，但它并不是一个期望获得大量收益的商业级游戏。相反，它只是一个示例，这个示例旨在说明本书中的技术、模式、方法、原理和代码，而且这个示例中使用的解决方案只是众多解决方案之一。

大家可以在 GitHub 的 WoF 仓库中看到 React + Flux UI，它位于一个后端服务之上，作为生态系统中其他微服务的代理。事件溯源和 CQRS 模式会展示游戏中的“移动操作”流如何产生可处理的事件流，事件的输出流通过实时消息传递被系统传递到活动的游戏中。

构建 WoF 有很多原因。首先，这个项目很有趣。其次，可以构建一个示例模板作为参考。目的不是让大家把它作为复制粘贴的源代码，而是帮助大家利用它来解决业务问题，启发我们使用 Go 语言来开发软件，这样做不仅富有成效、功能强大，而且还令人愉快，无论是用来解决复杂的业务问题还是构建实时的基于 Web 的 RPG（角色扮演游戏）。

15 结论

如果用桥梁建筑与编程类比，那么建设到一半的时候可能会发现，对岸比预想的要远 50 米，而且其材质是花岗岩而不是泥土，更糟糕的是我们最终想要的是一座公路桥而不是步行桥。

——Sam Newman，*Building Microservices* 作者

恭喜大家读完了两个夸夸其谈的作者的文章，听我们信口开河，编译书中的代码并运行了这些示例。读完这本书后，希望大家在用它给壁炉引火或作为鸟笼衬底前，能够学到了一些东西。

最后这章将介绍以下内容。

- 回顾整本书中提到的内容。
- 云原生 Go 开发学习和成长的后续步骤和建议。

我们学到了什么

希望大家读完这本书后能够有所收获。尽管没有徽章、升级和经验值奖励，但在把这本书丢到一边前，还是能收获一些有用的知识或信息的。

Go 不是小众语言

互联网圈子里有一种恶意刻板的印象，Go 语言虽然方便，但它的真正使用场景是建立命令行工具和一次性实用小程序。我们用简单优雅的代码解决复杂问题来反驳这种诽谤，相信大家现在应该知道 Go 语言的真正力量了。

微服务应该有多“微”

在构建不同复杂性的示例时，已经提到了该在何处拆分微服务以及用到的模式和技术。

虽然没有固定的答案，但是大家脑海中应该有几个架构设计的指导原则。

- 一个微服务应该只做一件事。它应该是一个独立的，基于 RESTful 单一责任原则（SRP）的实例。很容易改动和升级，改动它不会影响到其他服务。
- 来自 Sam Newman 书中的黄金法则：可以在改动一个服务并自己部署它的同时不改动任何其他东西吗？

持续交付和部署

如果对自己的应用程序没有信心，那么世界上所有的工具和技术都无法帮助你。如果不知道将程序部署在生产环境中表现如何，那么云将是你的敌人，而不是最好的盟友。

每次代码提交后应该在所有分支上尽快运行构建。构建流水线应该立即执行单元测试，并且尽快执行集成测试。

假设一切测试通过，构建流水线应该自动将应用程序部署到各种环境中。至少应该自动部署到测试环境中，然后使用一键部署来实现在更高级别的环境（如演示和生产环境）中进行部署。

测试一切

信心只来自测试。我们只有严格遵守测试驱动开发（TDD）的原则，才能对代码的可靠性和设计能力有信心。集成测试应在衔接处验证代码，验收测试应在独立环境中验证应用程序的所有部分。

许多人认为编程语言的选择是解决所有问题的银弹。如果能有一件事情解决大部分的问题，那就是测试。这其实很难，也需要努力，测试需要花费更多的时间，但这是维护现有应用程序、快速构建和部署新功能、通向成功的必经之路。

尽早发布，频繁发布

对许多企业和组织来说，恐惧发布是一个很真实的事情，即使他们将自己标榜为敏捷或精益创业。这些组织经常推迟发布直到最后一分钟，或者每季度甚至每 6

个月才安排一次发布。

要想降低新版本发布时的恐惧，最好的的方法不是减少发布次数，而是更频繁地发布。当功能经过测试和验证通过后，新版本就应该被发布出来并可用。可以使用蓝/绿部署等技术将这些功能逐步提供给消费应用程序的客户。这个模型被很多大公司采用，包括谷歌和 Facebook。

自动发布是必不可少的，版本发布频率与维护和增强应用程序的能力直接相关。正如我们在本书中提到的，应用程序的生命周期处于非“生产”阶段的时间是非常短的。要考虑到当应用程序正在被数百万人使用时如何在满足快速和可靠性的情况下交付应用，必须要为此做一个长远的规划。

事件溯源、CQRS 和更多首字母缩略词

我们知道，语言和技术不能代替可扩展的架构和设计。可以利用所谓的云原生语言来实现可怕的模式并彻底破坏应用程序的扩展能力。

本书讨论了支持大规模操作的模式和实践，包括事件溯源、CQRS（命令查询责任分离）模式和最终一致性。了解它们的工作原理以及何时使用其复杂的功能，这样能够帮助我们创建真正的云扩展应用程序。

下一步

现在大家已经成为一个使用 Go 构建云服务的专家，接下来将何去何从？

构建应用，无论什么应用。如果不使用本书中介绍的技能和技巧，那么大家将会很快忘记它们，这样会让我们感到沮丧难过。提出疯狂的想法并实现它们，构建应用应该像使用它们一样令人愉快。

做贡献，回馈社区，为其他构建微服务的 Go 开发人员编写代码库。创建示例，添加到本书的示例存储库中。我们给社区贡献的代码越多，社区就越强大，人们停止低估强大、灵活和优雅的 Go 语言的那天就会越早到来。

分享。把这本书告诉对 Go 持怀疑态度的朋友。送他们这本书，向他们介绍示例中的代码。宣传这本书，让更多的人痴迷于使用 Go 语言编写云原生代码。

附录 A
云应用的故障排查

如果我有一个小时来解决问题，我会用 55 分钟来思考这个问题，用 5 分钟来确定解决方案。

——阿尔伯特·爱因斯坦

如果大家从头到尾读完这本书，很有可能已经拥有了几个在云中部署和运行的应用程序示例。那么，下一个项目是什么？是要将一些旧代码迁移到云上，开启一个新的项目，还是一次性接管整个微服务？

无论什么项目，有时候你会发现自己正在查看显示器，观察已经部署并正在运行的云应用程序。

本附录将讨论在编写完应用程序后对于可能发生的问题进行故障排除的概念和技术。我们经常假定，应用程序编译时故障排除工作就已经完成，但其实这是不可能的。我们要一直维护这个程序直到它们被替换，所以应该知道如何在云上监控、评估和诊断它们。

使用日志流

真正的云原生应用程序不会将日志写到磁盘上。相反，它们会将日志流发送到 STDOUT 或 STDERRE 上。在云中运行的应用程序不应该知道日志的最终目的地和消费者，它们唯一需要关心的是将正确的信息发送到日志流中，以帮助监控和诊断（当然不会暴露敏感信息）。

如果使用 Pivotal Web Services 或者企业中安装了 Pivotal Cloud Foundry，则可以使用 `cf logs` 命令检查应用程序的日志。这可能对实时监视有用，但使用此命令看

到的日志不会永远持续输出。因为云中应用的日志传输数据量是非常惊人的，所以大多数云只提供实时监控和非常短的历史记录。

当想查看应用程序出了什么问题时，首先应该查看的就是日志流。无论是查看实时数据还是存储在 Splunk、ELK 堆栈、Sumologic 或其他工具中的历史数据，日志流都是不二之选。

应用程序发出的日志流应该被视为来自卫星轨道的遥测流。不能伸手触摸应用程序，不能拉出一个扳手或锤子来修复它。它在云端，可以从一个海岸移动到另一个海岸，切换到另一个大陆，或从一个示例变为一百个。在任何情况下，唯一保持不变的是发出的日志事件流。

这里隐含的假设是，应用程序实际上发出的都是有用的信息。虽然平台仍然可以使用计数器、指标和来自路由器的详细消息来丰富日志消息，但是应用程序本身仍然是决定日志需要记录哪些信息的关键。

健康和性能监控

正如我们在本书前面提到的，一旦应用程序被部署，它更应该像是一个被发射到太空的探测器，而不是像用电缆绑在我们身上的固定物体。可见性对生产中的应用程序非常重要，特别是对在云中部署的具有独立缩放组件的微服务生态系统应用程序来说。

我们需要能够一目了然地看到系统的整体运行状况。当需要更多详细信息时能够查看每个组件的运行状况和性能。如果没有这个，就不能支撑应用程序或承诺消费者满足“5 个 9”的正常运行时间。

应用性能监控（APM）工具

目前有无数的工具可以监控应用程序。需要知道每个服务的平均吞吐量，需要知道组件何时陷入困境，或者何时需要通过更改配置文件来进行扩展以适应新的需求或峰值事件。

New Relic（https://newrelic.com）就是此类工具之一。这种工具的工作方式是让应用程序向某种形式的中央聚合器发送性能指标。然后此聚合器负责存储和处理从应用程序中接收的信息，向管理员、开发人员和操作人员提供仪表板和下拉菜单。

许多像 New Relic 这样的 APM 解决方案都使用代理来运行。它们或是伴随着应用程序加载的库，或是伴随着通过“sidecar”样式加载的进程，这个进程有助于提高发送 APM 指标的量。当 New Relic 的页面不能链接到 Go 代理时，可以在以下链接中找到一个 OSS：https://github.com/yvasiyarov/gorelic。

不管应用程序是什么类型或有多复杂，当部署到生产环境时，如果没有强大的日志聚合和 APM 的能力，那么当现实环境出现问题时，只能盲目地依靠希望和猜测来排查问题。我们没说 New Relic 就是唯一的解决方案，只是大家应该找到一种适用于任何合理规模的生产应用程序的 APM 解决方案。

通过平台监控应用程序

APM 工具能够帮助我们跟踪方法调用的时间指标、REST API 的延迟和吞吐量以及许多其他指标。但是，平台还应该能够获取应用程序的重要指标和它们在平台级别上的表现。

例如，任何正规的云提供商都会提供访问原始数据或仪表板 GUI（最好是两者都有）的权限，显示平台统计的概览和详细信息。

平台应该向应用程序开发人员和运维人员公开的指标如下。

- 每个应用程序示例的 CPU 使用率。
- 每个应用程序示例的内存消耗。
- 每个应用示例的每个请求的响应时间。

如果平台不能提供这些基本信息，那么可能需要考虑换一个提供商。如果不能随时获取这些信息，那么维护这样一个生产应用程序就会像带着枷锁一样困难。

在云中调试应用程序

很多时候，本书都鼓励大家把应用程序看作部署的科学仪器，像在轨道上绕地球运行的卫星。这些应用程序不能被触及，当然也没有可以进行低级别通信的线缆。

这给那些认为无法使用调试器就不能完成故障排除的人们提出了一个问题。

如果本地工作站上运行着应用程序的本地副本，那么肯定可以使用 Go 调试器（https://golang.org/doc/gdb）。但是对于生产环境中跨越多个可用区域的多个实例部署应用程序，使用调试器不会带来任何好处。

即使实际上可以远程连接调试器，但仍需要找出要调试的示例。在许多情况下，问题可能分布在应用程序的多个示例上。例如，请求可能由示例 1 处理，来自相同用户的后续请求可以由示例 2 处理。假设两个应用程序都是真正的云原生程序，它们是无状态的，那么这些实例甚至可能不在同一个物理数据中心里面。

在无状态应用程序中，调用堆栈或本地查看中的信息可以帮助我们诊断问题。通过在云中的微服务对事务流、日志和事件流监控进行分析，通常比在服务的进程空间中窥视显得更有用。

不依赖交互式调试器就如同让许多人放弃安全毯一样。但越快拥抱其他可用于诊断云中问题的工具便越好。

假设应用程序是一颗运行在轨道上的卫星，我们是地面上的开发商。那么我们需要通过遥测（日志和诊断端点）获取应用程序的哪些信息，才可以诊断运行着的生产应用程序中的问题呢？

Broadview®
www.broadview.com.cn
博文视点·IT出版旗舰品牌
技术凝聚实力·专业创新出版

构建云服务的完整指南

《Cloud Native Go：构建基于Go和React的云原生Web应用与微服务》向开发人员展示了如何构建大规模云应用程序，在满足当今客户强大需求的同时还可以动态扩展处理几乎任何规模的数据量、流量或用户。

Kevin Hoffman和Dan Nemeth在书中详细描述了现代云原生应用程序，阐明了与快速、可靠的云原生开发相关的因素、规则和习惯。他们还介绍了Go这种“简单优雅”的高性能语言，它特别适用于云开发。

在本书中，我们将使用Go语言创建微服务，使用ReactJS和Flux添加前端Web组件，并学习基于Go的高级云原生技术。Hoffman和Nemeth向大家展示了如何使用Wercker、Docker和Dockerhub等工具构建持续交付流水线，自动推送应用程序到平台上，并系统地监控生产中应用程序的性能。

◎学习“云之道”：为什么开发出好的云软件的基本在于心态和规则

◎了解为什么Go语言是云本地微服务开发的理想选择

◎规划支持持续交付和部署的云应用程序

◎设计服务生态系统，然后以test-first方式构建它们

◎将正在进行的工作推送到云上

◎使用事件溯源和CQRS模式来响应大规模和高吞吐量

◎构建安全的基于云的Web应用程序

◎使用第三方消息传递供应商来创建响应式云应用程序

◎使用React和Flux构建大规模、云友好的GUI

◎监控云中的动态扩展、故障转移和容错

作者简介

Kevin Hoffman目前专门帮助企业将应用程序迁移到云端。他已构建的应用程序包括无人机远程控制程序、生物安全识别程序、超低延迟金融应用程序、移动应用程序等。他在为Pivotal Cloud Foundry创建自定义组件的时候爱上了Go语言。Dan Nemeth是Pivotal的咨询解决方案架构师，他从1995年开始使用C语言为当地的ISP编写专业的CGI脚本代码。他职业生涯的大部分时间都以独立顾问的身份为财经和制药行业提供解决方案，现在他正在专攻Go语言。

本书支持网站：informit.com/register
访问该网站可以下载本书资源，提交勘误

Pearson
www.pearson.com

策划编辑：孙奇俏
责任编辑：徐津平
封面设计：李　玲

上架建议：云计算

ISBN 978-7-121-32109-2

9 787121 321092 >

定价：69.00元